Backdoors in Satisfiability Problems

by

Zijie Li

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2009

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Zijie Li

Abstract

Although satisfiability problems (SAT) are NP-complete, state-of-the-art SAT solvers are able to solve large practical instances. The notion of backdoors has been introduced to capture structural properties of instances. Backdoors are a set of variables for which there exists some value assignment that leads to a polynomial-time solvable sub-problem. I address in this thesis the problem of finding all minimal backdoors, which is essential for studying value and variable ordering mistakes. I discuss our definition of sub-solvers and propose algorithms for finding backdoors. I implement our proposed algorithms by modifying a state-of-the-art SAT solver, Minisat. I analyze experimental results comparing our proposed algorithms to previous algorithms applied to random 3SAT, structured, and real-world instances. Our proposed algorithms improve over previous algorithms for finding backdoors in two ways. First, our algorithms often find smaller backdoors. Second, our algorithms often find a much larger number of backdoors.

Acknowledgments

I would like to thank my supervisor, Peter van Beek, for his guidance and support throughout my graduate studies. I truly appreciate his supervision of my research and his advice on my thesis. I would like to thank my readers, Robin Cohen and Chrysanne DiMarco, for their time and suggestions. I would like to thank my writing tutor, Nicole Keshav, for her help in improving my writing skills. I would like to thank Wei Li for discussing the problems with me. I would also like to thank my family for their constant support and for believing in me.

I am grateful to the School of Computer Science, the University of Waterloo, and Sharcnet for making the work presented in this thesis possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, I informally introduce and motivate the problem to be addressed in this thesis. I summarize the contributions of the thesis and give an outline of the rest of the thesis.

1.1 Motivation

The satisfiability problem (SAT) is a constraint satisfaction problem (CSP) where variables have Boolean domains and constraints are Boolean formulas. A formula in conjunctive normal form (CNF) is a conjunction of a finite set of clauses, where a clause is a disjunction of a finite set of literals. A literal is a Boolean variable or its negation. SAT is used for solving combinatorial problems, such as scheduling, planning, hardware and software verification.

SAT is the first known NP-complete problem. However, there exist polynomial-time tractable classes, such as 2SAT and Horn. A formula is 2SAT if every clause has at most two literals. A formula is Horn if every clause has at most one positive literal. Schaefer [26] proposes a dichotomy theorem that characterizes any SAT instance as either polynomial-time decidable or NP-complete.

Complete SAT solvers typically apply the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [3, 2], which enhances a backtracking algorithm with unit propagation and pure literal elimination. A backtracking algorithm explores the search tree using depth-first search. If a clause contains a single literal, unit propagation assigns the corresponding variable a satisfying value and simplifies the formula. If a literal occurs in a formula only positively or negatively, pure literal elimination assigns the corresponding variable a satisfying value and simplifies the formula.

Although SAT is NP-complete in theory, state-of-the-art SAT solvers scale well on large problem instances in practice. To explain the gap between theory and practice, Williams, Gomes, and Selman [29] introduce the notion of backdoors. Backdoors are a set of variables such that once the values of the variables are set properly, the simplified sub-problem can be solved by a sub-solver. Given a problem instance, a sub-solver either rejects the instance, or determines the satisfiability of the instance in polynomial time.

The problem that I address in this thesis is finding all minimal backdoors in SAT instances. A minimal backdoor is a backdoor with the smallest size in an instance. This problem is important for studying the problem hardness, which is generally represented as the time used or the number of nodes extended by a SAT solver. A problem instance can be solved by performing a backtracking search on backdoor variables. Thus, an instance with a large number of variables and clauses could be solved efficiently if it has small backdoors. In addition, identifying all minimal backdoors is a first step to investigating value and variable ordering mistakes. A variable ordering heuristic can make a mistake by selecting a variable not in the backdoor. A value ordering heuristic can make a mistake by assigning the backdoor variable a value that does not lead to a polynomial sub-problem. We are interested in studying how value and variable ordering mistakes affect the performance of backtracking algorithms.

1.2 Contributions of the Thesis

- We define sub-solvers both algorithmically and syntactically. Backdoors are defined with respect to sub-solvers. Algorithmically defined sub-solvers are polynomial-time techniques of current SAT solvers, such as unit propagation and pure literal elimination. Syntactically defined sub-solvers are polynomial-time tractable classes, such as 2SAT and Horn. Without considering the effect of unit propagation, the size of backdoors with respect to syntactically defined sub-solvers is relatively large. On the other hand, it is possible that the simplified sub-problem is polynomial-time solvable before algorithmically defined sub-solvers find a solution. Therefore, we propose sub-solvers that first apply unit propagation, and then checks polynomial-time decidable classes.

- We propose both systematic and local search algorithms for finding backdoors. The systematic search algorithm is guaranteed to find all minimal backdoors. However, it is unable to handle large instances because of its exponential complexity. Kilby, Slaney, Thiébaux, and Walsh [16] propose a local search algorithm to approximate a minimal backdoor, but our goal is to find as many minimal backdoors as possible. Hence, based on Kilby *et al.*'s algorithm, we propose two local search algorithms for finding small backdoors. Our first algorithm incorporates our definition of sub-solvers with Kilby *et al.*'s algorithm. Our second algorithm applies several local search techniques, such as Tabu Search, on finding small backdoors.

- We implement the algorithms for finding backdoors by modifying one of the fastest SAT solvers, Minisat [9]. A DPLL-based SAT solver, Satz-rand [10] is used as the sub-solver in previous work [29, 24, 16, 6]. However, Satz-rand is unable to efficiently solve large real-world instances. Minisat is able to handle large instances because it applies modern techniques, such as conflict-clause recording, in the search for solutions.

- We find small backdoors in large real-world instances. We are interested in

real-world instances because of their practical use. The test instances used in previous work are small instances that can be quickly solved by Minisat. We experiment on large real-world instances, especially the instances from SAT-Race 2008, to compare our proposed algorithms to previous algorithms.

Algorithms based on our proposed sub-solvers can find smaller backdoors than previous algorithms. Our proposed local search algorithms can find a significantly larger number of backdoors than previous algorithms. Although backdoors in large real-world instances consist of hundreds of variables, the backdoor size is notably small compared to the total number of variables. The results of our work can be used to study the problem hardness of large instances, and to study value and variable ordering mistakes.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. Chapter 2 introduces the concepts and definitions that are fundamental for understanding the work presented in this thesis. Chapter 3 reviews the approaches and results of previous research on backdoors. Chapter 4 discusses the systematic and local search algorithms for finding backdoors in SAT instances. Chapter 5 presents experimental results on random 3SAT, structured, and real-world instances, comparing our proposed algorithms for finding backdoors to previous algorithms. Chapter 6 concludes this thesis and suggests what could be done in the future.

Chapter 2

Background

In this chapter, I briefly review the necessary background in constraint programming, SAT solvers based on backtracking search, and local search algorithms. Then I formally define the problem addressed in this thesis. (For more background on these topics, see Rossi, van Beek, and Walsh [23], Mitchell [20], Hoos and Stützle [13], Williams, Gomes, and Selman [29], Schaefer [26], and Dechter [4].)

2.1 Constraint Programming

The constraint satisfaction problem (CSP) has a set of variables, each taking a value in a given domain, and a collection of constraints that specify the allowed combinations of values for some subsets of variables. A solution to a CSP is a value assignment to each variable that satisfies all of the constraints. A CSP is unsatisfiable if no solutions exist. The following definition is from [23].

Definition 2.1 (Constraint Satisfaction Problem (CSP))**.** *A CSP is a triple* $P = \langle X, D, C\rangle$ *where* X *is an* n*-tuple of variables* $X = \langle x_0, x_1, \ldots, x_{n-1}\rangle$*,* D *is a corresponding* n*-tuple of domains* $D = \langle D_0, D_1, \ldots, D_{n-1}\rangle$ *such that* $x_i \in D_i$*,* C *is a* t*-tuple of constraints* $C = \langle C_0, C_1, \ldots, C_{t-1}\rangle$*.*

The satisfiability problem (SAT) is a CSP where the domains of variables are Boolean values and the constraints are Boolean formulas. A formula in conjunctive normal form (CNF) is a conjunction of a finite set of clauses. Each clause is a disjunction of a finite set of literals. A literal is a Boolean variable x with an assigned parity $\varepsilon \in \{0, 1\}$, denoted by x^ε. A positive literal is denoted by $x = x^1$ and a negative literal is denoted by $\neg x = x^0$. A solution to a formula is a truth value assignment to each variable that satisfies all the clauses. Given a formula $F = \langle X, D, C\rangle$, we denote by $a_S : S \to D$ the partial value assignment to a subset of variables $S \subseteq X$. We define the simplified formula $F[a_s]$ as the formula after removing from F all the clauses that are satisfied by a_s, and removing from every clause all the literals that are false by a_s. A formula F contains an empty clause if F has both unit clauses $\{x\}$ and $\{\neg x\}$.

A clause is 2SAT if it contains at most two literals, and a formula is 2SAT if every clause is 2SAT. A clause is Horn if it has at most one positive literal, and a formula

is Horn if every clause is Horn. A formula is renamable Horn (RHorn) if it can be transformed into Horn by a uniform renaming of variables. Renaming a variable x is to replace every occurrence of x^{ε} with $x^{1-\varepsilon}$. Dechter [4] calls a clause anti-Horn if it has at most one negative literal, and a formula is anti-Horn if every clause is anti-Horn.

Example 2.1. *Examples of formulas*

CNF formula	$(x_0 \vee x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_4) \wedge (\neg x_0 \vee x_3 \vee x_5)$
2SAT	$(x_0 \vee x_1) \wedge (\neg x_1 \vee x_3) \wedge (x_0 \vee \neg x_2)$
Horn	$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4) \wedge (\neg x_0 \vee \neg x_2)$
Renamable Horn	$(x_0 \vee x_1) \wedge (x_2 \vee x_3)$ *By renaming* x_0 *to* $\neg x_0^*$ *and* x_2 *to* $\neg x_2^*$*, the formula is transformed into:* $(\neg x_0^* \vee x_1) \wedge (\neg x_2^* \vee x_3)$.
Anti-Horn	$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee x_2 \vee x_3)$

2.2 Features of SAT Solvers

The backtracking search is a fundamental algorithm for solving CSPs. A backtracking algorithm traverses the search tree of a problem instance in a depth-first manner. Every internal node of the search tree generated by backtracking search is a branching variable x. The edges from x to its children represent assigning to x possible values from the domain D. The path from the root to a node is a partial value assignment to the variables along the path. Value and variable ordering heuristics are used to guide the search. When the backtracking algorithm attempts to extend a node in the search tree, the variable ordering heuristic chooses a variable to be branched on and the value ordering heuristic decides the next value assigned to the variable. The performance of the backtracking algorithm can vary dramatically depending on value and variable ordering heuristics. The search tree of SAT instances is a binary tree, where left and right branches represent assigning the Boolean values $\{0\,(false), 1\,(true)\}$ to the branching variable.

A SAT solver based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [3, 2] applies unit propagation and pure literal elimination during the backtracking search. A unit clause has only one literal, so the value of its corresponding variable is determined. Unit propagation means if a formula contains unit clauses, the variables of the unit clauses are assigned the satisfying values. The value assignments are then propagated to simplify the formula. A pure literal is a literal x^{ε} such that $x^{1-\varepsilon}$ does not occur in the formula. Pure literal elimination means if a formula has pure literals, the corresponding variables are assigned the satisfying values to simplify the formula. Clause learning adds implied clauses deduced from the conflicts during unit propagation.

Example 2.2. *Consider a formula* $F_1 = x_0 \wedge (\neg x_0 \vee \neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$. *Notice* $\{x_0\}$ *is a unit clause, so we set* $x_0 = 1$. *Unit propagation implies* $x_1 = 0$ *and* F_1 *is simplified to* $(x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$.

Consider a formula $F_2 = (x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_1 \vee x_3)$. *Notice* x_3 *is a pure literal, so we set* $x_3 = 1$ *and* F_2 *is simplified to* $(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_1 \vee \neg x_2)$.

Given a formula F and a partial assignment a_s to a set of variables S, $F[a_s]$ denotes the simplified formula under a_s. We use the notation $a_s \cup \{l\}$ for the extension of a_s by the literal l. Mitchell [20] presents the DPLL algorithm as follows.

Algorithm DPLL(F, a_s)

SAT	**if** $F[a_s]$ is empty, **return** SATISFIABLE
Conflict	**if** $F[a_s]$ contains an empty clause, **return** UNSATISFIABLE
Unit Clause	**if** $F[a_s]$ contains a unit clause $\{p\}$, **return** DPLL$(F, a_s \cup \{p\})$
Pure Literal	**if** $F[a_s]$ contains a pure literal p, **return** DPLL$(F, a_s \cup \{p\})$
Branch	Let p be a literal of $F[a_s]$ **if** DPLL$(F, a_s \cup \{p\})$ returns SATISFIABLE, **return** SATISFIABLE **else return** DPLL$(F, a_s \cup \{\neg p\})$

Satz [19] is a DPLL-based SAT solver. A randomized version of Satz, Satz-rand [10], is used as the sub-solver in some previous work [29, 16, 24, 6] because Satz-rand has an effective variable selection heuristic and a strong simplification strategy. However, Satz-rand is unable to efficiently solve large real-world problem instances.

Based on the DPLL algorithm, Minisat [9] includes the techniques of conflict-clause recording, conflict-driven backjumping, dynamic variable ordering, and two-literal watch scheme. The version 2 of Minisat has a variable elimination style simplification. Algorithm 2.1 shows the search procedure of Minisat [9].

Algorithm 2.1: The search procedure of Minisat

```
while true do
    Propagate unit clauses;                    /* two-literal watch scheme */
    if no conflict then
        if all variables assigned then
            return SATISFIABLE;
        else
            Choose a new variable and assign it a value;
            /* dynamic variable ordering based on activity          */
    else
        Analyze and add a conflict clause; /* conflict-clause recording */
        if top-level conflict found then
            return UNSATISFIABLE;
        else
            Backtrack;                    /* conflict-driven backjumping */
```

We modify Minisat instead of Satz-rand because Minisat is one of the fastest SAT solvers. Minisat is highly efficient in solving large real-world instances used in SAT competitions. In addition, the code of Minisat is clear and well-documented.

2.3 Stochastic Local Search Algorithms

The definitions and algorithms in this section are taken from [13, 23].

The backtracking search algorithm is one of the systematic search algorithms, which are complete traversals over the search space. If there exists any solution to a problem instance, systematic search algorithms are guaranteed to find a solution. Otherwise, systematic search algorithms conclude the fact that the problem instance has no solutions. In contrast, local search algorithms are neither guaranteed to find a solution to the problem instance, nor able to firmly determine that no solution exists. Despite the incompleteness, local search algorithms are essential to solving large CSP instances.

Hoos and Stützle [13] have formally defined a stochastic local search (SLS) algorithm.

Definition 2.2 (Stochastic Local Search (SLS) Algorithm)**.** *Given a (combinatorial) problem* Π*, a* stochastic local search algorithm *for solving an arbitrary problem instance* $\pi \in \Pi$ *is defined by the following components:*

- *the* search space $S(\pi)$ *of instance* π*, which is a finite set of candidate solutions* $s \in S$ *(also called* search positions, locations, configurations, *or* states*);*
- *a* set of (feasible) solutions $S'(\pi) \subseteq S(\pi)$*;*
- *a* neighborhood relation *on* $S(\pi)$*,* $N(\pi) \subseteq S(\pi) \times S(\pi)$*;*
- *a finite* set of memory states $M(\pi)$*, which, in the case of SLS algorithms that do not use memory, may consist of a single state only;*
- *an* initialization function $init(\pi) : \emptyset \rightarrow \mathcal{D}(S(\pi) \times M(\pi))$*, which specifies a probability distribution over initial search positions and memory states;*
- *a* step function $step(\pi) : S(\pi) \times M(\pi) \rightarrow \mathcal{D}(S(\pi) \times M(\pi))$ *mapping each search position and memory state onto a probability distribution over its neighboring search positions and memory states;*
- *a* termination predicate $terminate(\pi) : S(\pi) \times M(\pi) \rightarrow \mathcal{D}(\{true, false\})$ *mapping each search position and memory state to a probability distribution over truth values, which indicates the probability with which the search is to be terminated upon reaching a specific point in the search space and memory state.*

In the above, $\mathcal{D}(S)$ *denotes the set of probability distributions over a given set* S*, where formally, a probability distribution* $D \in \mathcal{D}(S)$ *is a function* $D : S \rightarrow \mathbb{R}_0^+$ *that maps elements of* S *to their respective probabilities.*

Definitions of the search space, the set of (feasible) solutions, and the neighborhood relation usually depend on the problem instance π. Once these three components are determined, definitions of the initialization and step functions can be independent from π. Therefore, various general SLS algorithms are developed. I mainly introduce the Iterative Improvement (II) and Tabu Search (TS) algorithms, which are relevant to our study. The Iterative Improvement algorithm, Algorithm 2.2, is a basic SLS algorithm.

Algorithm 2.2: Iterative Improvement (II)

Determine initial candidate solution s;
while *s is not a local minimum* **do**
    Choose a neighbor $s' \in N(s)$ such that $g(s') < g(s)$;
    $s \leftarrow s'$;

The function $g(s) : S(\pi) \rightarrow \mathbb{R}$ is an evaluation function, which maps each search state s to a real number such that the global optima correspond to the (optimal) solutions of the problem instance π. The evaluation function directs the search towards a promising area that is likely to have solutions. Given a search space S, one common initialization function returns a uniform distribution over S, i.e., $init(s) \leftarrow 1/|S|$ if $s \in S$. Thus, an initial candidate solution s is selected randomly from the search space S. Then, the Iterative Improvement algorithm applies $g(s)$ to evaluate the neighborhood $N(s)$ of the current search state s. Use $I(s)$ to denote the set of neighbors $s' \in N(s)$ such that $g(s') < g(s)$. The step function can then return a uniform distribution over $I(s)$, i.e., $step(s)(s') \leftarrow 1/|I(s)|$ if $s' \in I(s)$; otherwise, $step(s)(s') \leftarrow 0$. Hence, a new candidate solution s' is chosen randomly from $I(s)$ to improve the current one s.

The Iterative Improvement algorithm terminates when it reaches a local minimum, which is defined as follows [13].

Definition 2.3 (Local Minimum)**.** *Given a search space S, a solution set $S' \subseteq S$, a neighborhood relation $N \subseteq S \times S$ and an evaluation function $g : S \rightarrow \mathbb{R}$, a* local minimum *is a candidate solution $s \in S$ such that for all $s' \in N(s)$, $g(s) \leq g(s')$.*

A search state s is a local minimum if s has no neighbors that can make an improvement regarding the evaluation function. To avoid being stuck in local minima, SLS algorithms are typically incorporated with escape strategies. Tabu Search, Algorithm 2.3, is one of the escape strategies. Tabu Search uses a short-term memory to record previously visited search states, which are forbidden to be revisited. As a result, Tabu Search allows the search to escape from local minima and avoids cycles in the search. The tabu tenure is a parameter that decides the number of search states to be memorized. However, it is possible that Tabu Search also excludes search states that lead to new candidate solutions.

Tabu Search allows non-improving moves. If $g(s') \geq g(s)$ for every non-tabu neighbor $s' \in N'(s)$, then Algorithm 2.3 selects a non-improving candidate solution s' in Line 1.

Algorithm 2.3: Tabu Search (TS)

Determine initial candidate solution s;
while *termination criterion is not satisfied* **do**
    Determine the set of non-tabu neighbors $N'(s) \subseteq N(s)$;
1   Choose a best candidate solution $s' \in N'(s)$;
    Update tabu attributes based on s';
    $s \leftarrow s'$;

2.4 Backdoors

Williams, Gomes, and Selman [29] formally define backbones, sub-solvers, weak backdoors, and strong backdoors. A CSP instance is denoted as $P = \langle X, D, C \rangle$ where X is the set of variables, D is the set of domains, and C is the set of constraints. For a subset of variables $S \subseteq X$, $a_S : S \to D$ denotes a partial value assignment to the variables in S. We use $P[a_S]$ to denote the simplified CSP instance obtained by removing from P every constraint that is satisfied by a_S. The notation $P[v/x]$ represents the simplified CSP instance after assigning the value v to the variable x.

Definition 2.4 (Backbone)**.** *S is a backbone if there is a unique partial assignment $a_S : S \to D$ such that $P[a_S]$ is satisfiable.*

A backbone is a subset of variables that have the same value assignment in all solutions, whereas a backdoor is defined with respect to a sub-solver.

Definition 2.5 (Sub-Solver)**.** *A sub-solver A given as input a CSP, P, satisfies the following:*

- *(Trichotomy) A either rejects the input P, or "determines" P correctly (as unsatisfiable or satisfiable, returning a solution if satisfiable),*
- *(Efficiency) A runs in polynomial time,*
- *(Trivial solvability) A can determine if P is trivially true (has no constraints) or trivially false (has a contradictory constraint),*
- *(Self-reducibility) if A determines P, then for any variable x and value v, then A determines $P[v/x]$.*

The DPLL algorithm can be modified to be a sub-solver. We use only unit propagation and pure literal elimination. The algorithm stops when a branching step is encountered.

Definition 2.6 (Weak Backdoor)**.** *A nonempty subset S of the variables is a weak backdoor in P for A if for some $a_S : S \to D$, A returns a satisfying assignment of $P[a_S]$.*

A weak backdoor is a subset of variables such that some value assignment leads to a polynomial-time solvable sub-problem. Here is an example of a weak backdoor with respect to unit propagation.

Example 2.3. *Consider a formula* $F = (x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_2 \vee \neg x_3 \vee x_4)$. *We select variable* x_0 *and set* $x_0 = 1$. *Unit propagation implies* $x_3 = 1$, $x_1 = 1$, $x_2 = 1$, *and* $x_4 = 1$. *Now every variable has a value assignment and the formula* F *is satisfied. Hence,* x_0 *is a weak backdoor in* F *with respect to unit propagation.*

Definition 2.7 (Strong Backdoor)**.** *A nonempty subset* S *of the variables is a strong backdoor in* P *for* A *if for all* $a_S : S \to D$, A *returns a satisfying assignment or concludes unsatisfiability of* $P[a_S]$.

A strong backdoor is a subset of variables such that every value assignment leads to a polynomial-time solvable sub-problem.

Example 2.4. *Consider the formula* $F = (x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_2 \vee \neg x_3 \vee x_4)$ *in the previous example. After we assign* x_0 *the value* 1, *unit propagation can find a satisfying assignment of* F. *If we set* $x_0 = 0$, *by unit propagation we have* $x_1 = 1$ *and* $x_2 = 1$. *The simplified formula is* $(\neg x_3 \vee x_4)$ *and* F *is not yet satisfied. Hence,* x_0 *is not a strong backdoor in* F *with respect to unit propagation.*

I will review previous work on deletion backdoors and learning-sensitive backdoors in Chapter 3, so I introduce the concepts here. Dilkina, Gomes, and Sabharwal [6, 7] define deletion backdoors and learning-sensitive backdoors for SAT instances. A formula is denoted as $F = \langle X, D, C \rangle$ where X is the set of variables, D is the set of Boolean domains, and C is the set of constraints. For a subset of variables $S \subseteq X$, $F - S$ denotes the formula obtained by deleting the variables in S from F.

Definition 2.8 (Deletion Backdoor)**.** *A nonempty subset* S *of the variables is a deletion backdoor in* F *for* A *if* A *returns a satisfying assignment or concludes unsatisfiability of* $F - S$.

A deletion backdoor is a subset of variables such that once the variables are deleted from the formula, the remaining sub-formula can be solved in polynomial time.

Example 2.5. *Consider a formula* $F = (x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2 \vee x_3)$. *After deleting the variable* x_3 *from* F, *the formula* $F - \{x_3\} = (x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_0 \vee \neg x_2)$ *is 2SAT. Thus,* x_3 *is a deletion backdoor in* F *with respect to 2SAT.*

Definition 2.9 (Learning-Sensitive Backdoor)**.** *A nonempty subset* S *of the variables is a learning-sensitive backdoor in* F *for* A *if there exists a search tree exploration order such that a clause learning SAT solver branching only on the variables in* S, *with this order and with* A *as the sub-solver at the leaves of the search tree, either finds a satisfying assignment or concludes unsatisfiability of* F.

Dilkina *et al.* provide an example of a learning-sensitive backdoor in [7].

2.5 Dichotomy Theorem

I introduce the dichotomy theorem in this section because we want to identify polynomial tractable classes for finding backdoors.

Schaefer [26] has proposed a dichotomy theorem that characterizes any SAT instance as either polynomial-time decidable or NP-complete.

Theorem 2.1. *A SAT problem over a finite set of logical relations is polynomial-time decidable if at least one of the following six conditions holds:*

1. *Every relation is satisfied when all variables are 0.*
2. *Every relation is satisfied when all variables are 1.*
3. *Every relation is definable by a CNF formula in which each conjunct has at most one negative literal.*
4. *Every relation is definable by a CNF formula in which each conjunct has at most one positive literal.*
5. *Every relation is definable by a CNF formula in which each conjunct has at most two literals.*
6. *Every relation is the set of solutions of a system of linear equation over the two-element field $\{0, 1\}$.*

Jeavons, Cohen, and Gyssens [15] extend Schaefer's results to CSP instances. They suggest any set of constraints that is not NP-complete must satisfy an algebraic closure property. Given a set of constraints, Jeavons *et al.* construct a CSP to determine the closure property of the constraints.

2.6 Summary

In this chapter, I described the background knowledge of CSP, SAT, local search algorithms, backdoors, and polynomial-time tractable classes.

In the next chapter, I will review the related work on finding backdoors in SAT instances.

Chapter 3

Related Work

In this chapter, I summarize previous research on finding weak and strong backdoors. I also review previous work on the relationship between backdoor variables and on generalizations of backdoors.

3.1 Backdoors

Even though CSP and SAT problems are generally NP-complete, the state-of-the-art CSP and SAT solvers manage to solve large practical problems with thousands of variables and constraints. To explain why current CSP and SAT solvers scale well in practice, Williams, Gomes, and Selman [29] propose the notion of backdoors to capture structural properties of the problems. We have introduced their definitions of sub-solvers, backdoors, and strong backdoors in Chapter 2. Backdoors are a set of variables for which there exists some value assignment that leads to a polynomial-time solvable sub-problem. Using Satz-rand as the sub-solver, Williams *et al.* empirically find backdoors in several structured SAT instances. Experimental results show that the size of backdoors in these practical instances is relatively small compared to the number of variables, which means practical instances usually have small tractable structures.

In addition, Williams *et al.* provide three algorithms to solve CSPs by first exploiting backdoor variables. Once a backdoor is found, the sub-solver can either return a satisfying assignment or conclude the unsatisfiability of the problem. The first algorithm is deterministic and systematically searches every subset of variables for a backdoor. I will describe and modify this algorithm to find minimal backdoors in Chapter 4. Based on a beforehand estimate of the backdoor size, the second algorithm employs a randomized strategy. In each iteration of the algorithm, a subset of variables, whose size is larger than the backdoor size, is selected randomly and tested for a backdoor. The third algorithm is a depth-first search algorithm with a variable selection heuristic that chooses a backdoor variable to branch on with a certain probability. Based on a formal runtime analysis of these algorithms, Williams *et al.* believe that when the size of backdoors is small, exploiting backdoors is a benefit to solving CSPs.

Williams, Gomes, and Selman [30] further investigate the relations between backdoors, restarts, and heavy-tailed behaviors. A restart strategy means when a backtracking algorithm fails to find a solution after some period, the algorithm restarts with different value and variable ordering heuristics [23]. The heavy-tailed behavior means that a backtracking algorithm will very likely run for a long time on the problem instance [23]. Williams *et al.* suggest that small strong backdoors lead to runtime distributions lower-bounded by heavy-tails.

Backdoors are determined with respect to specific sub-solvers, which can be polynomial tractable syntactic classes, or polynomial algorithmic techniques of current solvers. Sub-solvers for SAT problems accept formulas as input. Because 2SAT, Horn, or RHorn formulas are well-known polynomial-time tractable, sub-solvers can be 2SAT, Horn, or RHorn solvers. State-of-the-art SAT solvers, such as Satz-rand, can also be modified to be sub-solvers because unit propagation and pure literal elimination are common polynomial-time features of SAT solvers.

Interian [14] proposes several approximation algorithms for computing backdoors in random 3SAT problems with respect to 2SAT and Horn. The basis of the first algorithm for 2SAT backdoors is finding a maximal independent clause set. A maximal independent clause set $\mathcal{C}$ is a set of clauses in a formula F such that no two clauses in $\mathcal{C}$ share variables, while every remaining clause of F has some variable appearing in $\mathcal{C}$. In fact, any value assignment to the variables in $\mathcal{C}$ leads to a simplified 2SAT formula. Thus, the variables in a maximal independent clause set form a 2SAT backdoor. Interian proves that this algorithm has an approximation ratio of 3, which means the size of backdoors found by this algorithm can be as large as three times the minimal size. The second algorithm calculates a 2SAT backdoor S by repeatedly adding variables to S. If a clause C shares no variables with the current backdoor S, a variable of C is added to S. A similar algorithm is used to find Horn backdoors. For each non-Horn clause C, the algorithm examines the positive literals whose corresponding variables are not in the current backdoor S. To make C become Horn, variables corresponding to the extra positive literals are added to S. Furthermore, Interian considers mapping the backdoor detection problem to the following hitting set problem.

HITTING SET	
Input	A collection $\mathcal{C}$ of subsets $C_1, \ldots, C_m$ of a finite set $V = \bigcup_{i=1}^{m} C_i$.
Parameter	A positive integer k.
Question	Is there a subset $V' \subseteq V$ of size at most k such that $V' \cap C_i \neq \emptyset$, $i = 1, \ldots, m$?

In the context of finding 2SAT backdoors, $\mathcal{C}$ is the set of all clauses, V is the set of all variables, and V' is the subset of variables forming a 2SAT backdoor. A mapping for finding Horn backdoors is also constructed in [14]. Interian analyzes the approximation ratio of the algorithms and indicates that a greedy algorithm for finding a minimum hitting set works best in practice. In addition, Interian concludes that compared to practical SAT problems, random 3SAT problems have larger backdoor sets, which include 30%–65% of the total number of variables. However, Interian fails to take into account the effect of unit propagation, which can result in much smaller backdoors.

The focus of some previous work [21, 28] is on the theoretical analysis of the parameterized complexity of the following two problems.

WEAK A-BACKDOOR	
Input	A formula F.
Parameter	A positive integer k.
Question	Does F have a weak backdoor of size at most k with respect to the sub-solver A?
STRONG A-BACKDOOR	
Input	A formula F.
Parameter	A positive integer k.
Question	Does F have a strong backdoor of size at most k with respect to the sub-solver A?

Szeider [28] analyzes the parameterized complexity of finding weak and strong backdoors with respect to DPLL sub-solvers. Applying only unit propagation or pure literal elimination or both, the DPLL sub-solver either returns a satisfying assignment of the given formula, or stops if branching is required. Szeider proves that the parameterized problems of weak and strong DPLL-backdoors are both W[P]-complete. (See Downey and Fellows [8] for more information on parameterized complexity.)

Nishimura, Ragde, and Szeider [21] investigate the parameterized complexity of weak and strong backdoor detection with respect to 2SAT and Horn. By reducing the hitting set problem to the weak 2SAT- or Horn-backdoor problem, they prove that the parameterized problem of detecting weak backdoors with respect to 2SAT and Horn is $W[2]$-hard, which is unlikely to be fixed-parameter tractable. Moreover, they propose two recursive algorithms, called `sb-2cnf` and `sb-horn` respectively, for finding strong 2SAT and Horn backdoors with size up to k. The algorithm `sb-2cnf` changes all clauses into 2SAT by recursively selecting a clause C with more than two variables. One variable of C is added into the backdoor set, and its occurrence in the formula is deleted. The algorithm `sb-horn` applies a similar idea to make non-Horn clauses into Horn. Based on the time complexity of these two algorithms, Nishimura *et al.* conclude that the parameterized problem of strong 2SAT- or Horn-backdoor is fixed-parameter tractable. They conjecture it is hard to detect weak backdoors because in addition to fulfilling the syntactic requirements, the simplified formula has to be satisfiable. Furthermore, they prove that the non-parameterized problems of both weak and strong 2SAT- or Horn-backdoor are NP-complete. However, as noted by Dilkina, Gomes, and Sabharwal [6], the algorithms proposed by Nishimura *et al.* fail to take into account any simplification caused by value assignments to backdoor variables. Therefore, the backdoors found by `sb-2cnf` and `sb-horn` are more properly called deletion backdoors.

Dilkina, Gomes, and Sabharwal [6] consider the influence of empty clause detection, a feature of state-of-the-art SAT solvers. A formula F contains an empty clause if F has both unit clauses $\{x\}$ and $\{\neg x\}$. They prove that the problem of strong 2SAT- or Horn-backdoor becomes both NP-hard and coNP-hard once the feature of empty clause detection is included. Despite increasing the worst-case complexity, the incorporation of empty clause detection can reduce the backdoor size in practice. For backdoors with respect to RHorn, they show that in some problems, strong backdoors can be exponentially smaller than deletion backdoors. Experimental results illustrate

that deletion RHorn backdoors have smaller sizes than strong Horn backdoors, although both have considerably larger sizes than strong backdoors with respect to DPLL sub-solvers and Satz-rand. Satz-rand performs best in finding minimal strong backdoors in their experiments on structured instances because Satz-rand solves the instances with little or no search.

Paris, Ostrowski, Siegel, and Saïs [22] introduce a two-step approach to find minimal strong RHorn backdoors. The first step is to find a renaming of variables that maximizes the number of Horn clauses in the formula. To approximate the maximum Horn sub-formula, they use a local search algorithm, which flips the value of a chosen variable in each iteration until one of the terminating criteria is met. In the second step, a greedy algorithm is applied to find a minimal backdoor. The algorithm repeatedly adds into the backdoor set a variable that appears in the most non-Horn clauses. Then, all corresponding positive literals of the chosen variable are deleted to make non-Horn clauses become Horn. Experimental results from random 3SAT and real-world SAT instances demonstrate that exploiting backdoor variables can significantly improve the performance of SAT solvers. However, backdoors found by the algorithm proposed by Paris *et al.* are more properly called deletion RHorn backdoors than strong RHorn backdoors. Moreover, they fail to explain what values are used in the experiments for the parameters of their proposed local search algorithm.

Kottler, Kaufmann, and Sinz [17] suggest two algorithms to identify deletion RHorn backdoors based on a transformation into the dependency graph problem. Given a formula F with n variables, the corresponding dependency graph is a directed graph with $2n$ vertices, where each variable is represented by two vertices k^0 and k^1, showing whether or not the variable should be renamed. A variable has a conflict loop if there is a path from k^0 to k^1, and vice versa. Kottler *et al.* provide the lemma: a formula is RHorn if and only if there exists no variable that has a conflict loop in the dependency graph. Thus, the main idea of their proposed algorithms is to break all conflict loops in the dependency graph. Both algorithms calculate small conflict loops in the dependency graph beforehand. If a variable is added into the backdoor set, its corresponding vertices and incident edges are removed from the dependency graph. The first algorithm adds variables to the backdoor set in a greedy way. The variable appearing in the most conflict loops is selected, and the process repeats until the remaining graph has no conflict loops. The second algorithm eliminates conflict loops by repeatedly selecting a conflict loop and adding all the involved variables into the backdoor set. If restoring the vertices and edges of a backdoor variable does not introduce new conflict loops into the dependency graph, the variable is deleted from the backdoor set to reduce the backdoor size. If the largest chosen conflict loop contains k variables, the size of backdoors found by this algorithm can be as large as k times the minimal size. Thus, the algorithm selects conflict loops with preference to the smallest ones. Empirical results show that the size of deletion RHorn backdoors in real-world SAT instances is usually smaller than the size of 2SAT or Horn backdoors.

Samer and Szeider [25] propose the concept of backdoor trees to investigate the relationship between variables of a strong backdoor set. A backdoor tree is a binary decision tree on variables that form a strong backdoor, whose leaves correspond to the

polynomial-time tractable sub-formulas the strong backdoor leads to. The following parameterized problem is discussed.

A-**BACKDOOR TREE**	
Input	A formula F.
Parameter	A positive integer k.
Question	Does F have a backdoor tree with at most k leaves with respect to the sub-solver A?

Samer and Szeider [25] syntactically define the sub-solver as the classes of 2SAT, Horn, and RHorn formulas. They prove that the problem of 2SAT- or Horn-backdoor tree is fixed-parameter tractable, while the non-parameterized problem of A-backdoor tree is NP-hard for $A \in \{\text{2SAT, Horn, RHorn}\}$. Moreover, they empirically compare the size of strong Horn backdoors, deletion RHorn backdoors, and Horn- or RHorn-backdoor trees in automotive configuration [27] and random 3SAT instances. They use the same set of automotive configuration instances as Dilkina *et al.* [6]. However, I will show in Chapter 5 that these automotive configuration instances are simple and mostly are solved by Minisat's pre-processing.

Ruan, Kautz, and Horvitz [24] study the connections between problem hardness and backdoors for quasigroup completion and morphed graph coloring problems. Using Satz-rand as the sub-solver in their experiments, they notice that problem hardness, represented by the runtime of Satz-rand, does not appear to be correlated with the size of backdoors. Therefore, the notion of backdoor keys is introduced to reveal the dependencies between backdoor variables. A backdoor key is a backdoor variable whose value is fixed given a value assignment to the rest of backdoor variables. Ruan *et al.* suggest that problem hardness has strong correlation with the ratio of the size of backdoor keys to the size of backdoors. Nonetheless, they also notice that no such correlation exists in domains such as logistics and circuit synthesis.

Kilby, Slaney, Thiébaux, and Walsh [16] theoretically prove the NP-hardness of backbone approximation. By modifying Satz-rand for the sub-solver, they propose a series of algorithms for finding both weak and strong backdoors. I will discuss their proposed algorithms later in Chapter 4. Experimental results from random 3SAT instances demonstrate that backbones do not overlap with backdoors very much. In addition, Kilby *et al.* empirically evaluate the correlations between backbones, backdoors, and problem hardness. They suggest that problem hardness, represented by the number of search nodes of Satz-rand, is most likely correlated with the size of strong backdoors. However, they only consider random 3SAT and graph coloring instances in the experiments, and these are small instances with at most 225 variables.

Extending the results in [16], Gregory, Fox, and Long [11] show that there is little overlap between backbones and backdoors for structured SAT instances. They propose an algorithm to find minimal backdoors by sequentially deleting unnecessary variables from the candidate backdoor. If the DPLL sub-solver can find a solution without branching, then the subset of variables is a backdoor. Experiments are carried out on both random 3SAT and structured instances, taken from domains including planning, graph coloring, and quasigroup completion. For a particular instance, Gregory *et al.* analyze the frequency that each variable appears in the 100 backdoors computed by their proposed algorithm. They also observe that backbone variables

are likely corresponding to backdoor variables with low frequencies. In addition, they indicate that combining clause learning with the sub-solver can reduce the size of backdoors. However, Gregory *et al.* only find 100 backdoors for each instance by repeatedly running their proposed algorithm. It is possible that the backdoors found do not well represent all the backdoors in an instance. Moreover, they only use small instances in their experiments with up to about 300 variables.

As a common feature of most state-of-the-art SAT solvers, clause learning can derive new clauses from the conflicts during propagation. Dilkina, Gomes, and Sabharwal [7] take clause learning into account and propose the idea of learning-sensitive backdoors. They prove that using unit propagation as the sub-solver, the minimal learning-sensitive backdoors can be smaller than the minimal traditional strong backdoors in some unsatisfiable SAT instances. For traditional strong backdoors, the order of value assignments to backdoor variables makes no difference. Nevertheless, the value ordering of variables in learning-sensitive backdoors is significant because for some unsatisfiable SAT instances, one value ordering can lead to a backdoor with size exponentially smaller than a different value ordering. Dilkina *et al.* empirically evaluate the upper bounds on the minimal size of learning-sensitive and traditional strong backdoors with respect to unit propagation. Experimental results from practical SAT instances suggest that clause learning leads to learning-sensitive backdoors with size considerably smaller than traditional strong backdoors.

Observing the similarities between SAT and Mixed Integer Programming (MIP), Dilkina *et al.* [5] introduce generalized backdoors for combinatorial optimization problems, including weak optimality backdoors for finding optimal solutions and optimality-proof backdoors for proving optimality. Analogous to the clause learning feature of SAT solvers, optimization algorithms add cuts and tightened bounds during the search. Hence, they also define order-sensitive backdoors for combinatorial optimization problems. They empirically analyze the lower bound on the probability that a randomly selected set of variables of a given size k is a backdoor. Experimental results show that benchmark instances in MIP tend to have small backdoors, and optimality-proof backdoors can be smaller than weak optimality backdoors. Moreover, the root Linear Programming (LP) relaxation is a good heuristic for discovering small backdoors.

3.2 Summary

In this chapter, I described the related work on both weak and strong backdoors. Experimental results in previous work show that backdoors with respect to syntactically defined sub-solvers tend to have larger sizes than backdoors with respect to algorithmically defined sub-solvers. Table 3.1 summarizes the sub-solvers and the domains of test instances used in the previous research.

In the next chapter, I will present several algorithms for finding backdoors.

Table 3.1: Summary of related work

	Sub-Solvers	**Instance domains**
Williams *et al.* [29]	Satz-rand	structured
Interian [14]	2SAT and Horn	random 3SAT
Szeider [28]	DPLL	
Nishimura *et al.* [21]	2SAT and Horn	
Dilkina *et al.* [6]	Horn, RHorn, DPLL, and Satz-rand	graph coloring, planning, game theory, and automotive configuration
Paris *et al.* [22]	RHorn	random 3SAT, SAT competition
Kottler *et al.* [17]	2SAT, Horn, and RHorn	SAT competition and automotive configuration
Samer and Szeider [25]	2SAT, Horn, and RHorn	automotive configuration
Ruan *et al.* [24]	Satz-rand	quasigroup completion and morphed graph coloring
Kilby *et al.* [16]	Satz-rand	random 3SAT
Gregory *et al.* [11]	DPLL	random 3SAT, planning, graph coloring, and quasigroup completion
Dilkina *et al.* [7]	unit propagation and clause learning	structured

Chapter 4

Algorithms for Finding Backdoors

In this chapter, I introduce how we define sub-solvers in our research and describe several algorithms for finding minimal backdoors. The ultimate goal of our research is to evaluate the effects on the runtime of backtracking algorithms when value and variable ordering heuristics make mistakes. Therefore, we address the problem of finding all or many minimal backdoors. First, I describe an exact algorithm, which is suitable for small and simple instances. Then, I explain the previous local search algorithm proposed by Kilby, Slaney, Thiébaux, and Walsh [16]. I also discuss a local search algorithm that combines our definition of sub-solvers with Kilby *et al.*'s algorithm. Last but not least, I analyze the techniques used in our proposed local search algorithm for improving Kilby *et al.*'s algorithm.

I introduce some terminology used in this chapter. The algorithms focus on finding weak backdoors in SAT instances, so if not explicitly specified, the word *backdoor* is used for weak backdoors. A *minimal backdoor* is a backdoor S such that for every backdoor S' in the instance, $|S| \leq |S'|$. A *small backdoor* refers to a backdoor S such that no proper subset of S is also a backdoor. A minimal backdoor can be viewed as a global minimum, and a small backdoor can be viewed as a local minimum. The *current smallest backdoor* is the smallest backdoor found so far by the algorithm.

I use the random 3SAT instance *uf50-01* as a running example (See Appendix A). Taken from SATLIB [12], the instance *uf50-01* is satisfiable with 50 variables and 218 clauses. I use subscripts $\langle 0, 1, \ldots, 49\rangle$ to represent variables $\langle x_0, x_1, \ldots, x_{49}\rangle$. For each variable x_i, $i = 0, 1, \ldots, 49$, I denote the positive literal as l_{2i} and the negative literal as l_{2i+1}.

4.1 Sub-Solvers

We apply the sub-solver and backdoor definitions in Chapter 2 to SAT. Given a formula F with n variables $X = \langle x_0, x_1, \ldots, x_{n-1}\rangle$, the domain D_i of each variable x_i is $D_i = \{0\,(false), 1\,(true)\}$, $i = 0, \ldots, n-1$. A nonempty subset S of the variables is a backdoor in F for a sub-solver A if for some partial assignment $a_S : S \to D$, A returns a satisfying assignment of the simplified formula $F[a_S]$.

We know from Chapter 3 that the sub-solvers in previous work are defined either

syntactically or algorithmically. Satz-rand can be modified to be a sub-solver. At each branching step, Satz-rand extends a variable chosen by the variable selection heuristic and assigns a truth value to the variable. Then, Satz-rand applies unit propagation and simplification, forcing some value assignments. The search continues until every variable has a satisfying value. A Satz-rand sub-solver applies only unit propagation and simplification. Given a simplified formula, a Satz-rand sub-solver either returns a satisfying assignment or stops when a branching step is encountered.

Kilby *et al.* [16] propose Algorithm 4.1 to compute small backdoor sets with respect to Satz-rand.

Algorithm 4.1: `MinWeakBackdoor`(F, I)

input : Formula F
A set of literals I forming a candidate backdoor
output: A set of literals W forming a small backdoor

$W \leftarrow \emptyset$;
while $I \neq \emptyset$ **do**
  Choose literal $l \in I$ sequentially;
  `/* randomized version: Choose literal` $l \in I$ `randomly; */`
  $I \leftarrow I \setminus \{l\}$;
  Run Satz-rand on $F[W \cup I]$;
  if Satz-rand *requires branching* **then**
    $W \leftarrow W \cup \{l\}$;
return W;

The input I is a candidate backdoor; that is, Satz-rand can solve $F[I]$ without branching. Initially all the literals in I form a backdoor, but we want to remove any unnecessary literals from I. If we remove a literal l and Satz-rand can still solve the simplified formula without branching, then the literal l is considered unnecessary for a small backdoor. The set W contains the literals that form a small backdoor. After removing a literal l from I, we run Satz-rand on F with a partial assignment $W \cup I$. If Satz-rand requires branching, then l belongs to the small backdoor set and is added to W.

However, the simplified formula can be polynomial-time solvable before Satz-rand returns a satisfying assignment. For example, it is possible that the simplified formula $F[a_S]$ is 2SAT, which belongs to the polynomial-time decidable classes, but unit propagation is unable to solve $F[a_S]$ without branching.

Therefore, in our proposed framework, we define the sub-solver both algorithmically and syntactically. We assume there exists a sub-solver that:

- applies unit propagation;
- solves the simplified formula if it is in one of the polynomial tractable classes.

Specifically, given a partial assignment a_S to a subset of variables S, we first apply unit propagation to obtain the simplified formula $F[a_S]$. Second, we check the

following conditions to see if $F[a_S]$ belongs to Schaefer's polynomial-time tractable classes described in Chapter 2:

1. if F is satisfied;
2. if $F[a_S]$ is 2SAT;
3. if F is satisfied after assigning 0 (*false*) to all the remaining variables;
4. if F is satisfied after assigning 1 (*true*) to all the remaining variables.

If one of the above conditions is true, then S is a backdoor set. The first two conditions are trivial. If F is satisfied, then a solution of F has been already found. $F[a_S]$ is 2SAT corresponds to Class 5 of Schaefer's theorem. The third condition covers Class 1 and 4 of Schaefer's theorem, while the last condition covers Class 2 and 3. If $F[a_S]$ is a Horn formula, it can be satisfied by assigning 0 to all variables unless it has unit clauses with a single positive literal. However, after unit propagation $F[a_S]$ is guaranteed to have at least two unassigned literals in each clause. In this case, at least one of the unassigned literals is a negative literal. Thus, $F[a_S]$ is satisfied when all remaining variables are 0, which means F is satisfied. A similar reasoning applies if $F[a_S]$ is an anti-Horn formula.

However, we do not cover the following polynomial-time tractable classes because of a lack of time:

- if $F[a_S]$ belongs to Class 6 of Schaefer's theorem;
- if $F[a_S]$ is RHorn.

Jeavons, Cohen, and Gyssens [15] note that Schaefer's theorem identifies the types of clauses that are closed under conjunction. The clauses can be put together with conjunction while still maintaining tractability. Jeavons *et al.* suggest that testing for tractability of any set of relations over a Boolean domain can be done in polynomial time by solving a CSP with eight variables. RHorn is not one of Schaefer's tractable classes because RHorn clauses do not satisfy the closure property. Any clause by itself is trivially RHorn, but an arbitrary conjunction of RHorn clauses is not guaranteed to be polynomial-time tractable. Nevertheless, determining whether a set of clauses is RHorn can be done in polynomial time. Given a formula F, Lewis [18] proposes a transformation of F into a 2SAT formula F^* and proves that F is RHorn if and only if F^* is satisfiable. Both constructing F^* and checking the satisfiability of F^* require polynomial time. Due to a lack of time, we did not implement algorithms for checking Class 6 of Schaefer's theorem and RHorn. Although we expect this does not significantly change our experimental results in Chapter 5, it is a matter for future experiments to decide. I will discuss this problem later in Chapter 6.

We have proposed three procedures, listed in Table 4.1, to check for the above mentioned conditions.

Table 4.1: Low-level procedures

Procedure	Input	Output	Description
`isSatisfied(`F`)`	Formula F	Return true if F is satisfied. Otherwise, return false.	Test if every clause of F is satisfied.
`is2SAT(`F`)`	Formula F	Return true if the remaining formula is 2SAT. Otherwise, return false.	For each not yet satisfied clause C of F, test if C has at most two unassigned variables.
`setValue(`F, v`)`	Formula F, Boolean value v	Return true if F is satisfied after assigning v to all the remaining variables. Otherwise, return false.	Assign v to each unassigned variable of F. Test if F is satisfied.

4.2 Exact Algorithm

Let k be the size of a minimal backdoor. Given a formula F with n variables $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$, we can perform an exhaustive search if k is small. There are $\binom{n}{k}$ subsets of k variables, each having 2^k possible value assignments. The time complexity is:

$$\sum_{i=1}^{k} \left(f(n)\, 2^i \binom{n}{i} \right) = O\left(f(n) 2^k n^k \right),$$

where $f(n)$ is the time complexity of the sub-solver A.

In fact, this brute force algorithm only works on very small instances. Even if the backdoor size k is small, the time complexity becomes infeasible in practice as n increases.

Algorithm 4.2 is used to find all the backdoors of size at most k. The algorithm calls a recursive procedure `expand(`V, S, k`)`, which explores the variables of F in a depth-first manner.

Algorithm 4.2: Exact algorithm for finding minimal backdoors

input : Formula F
Minimal backdoor size k
output: A set of minimal backdoors S of size k; or the empty set if F has no backdoors of size at most k

$S \leftarrow \emptyset$;
for $i \leftarrow 0$ **to** $n-1$ **do** `expand(`$\{x_i\}$, S, k`)`;
return S;

The procedure `expand(`V, S, k`)` takes as input a set of variables V, a set of minimal backdoors S, and a positive integer k. The integer k is used to control the levels of recursion. The procedure propagates every value assignment to the variables in V. If there is no conflict during propagation, we check if the formula F is satisfied or if the simplified formula is in one of the polynomial-time tractable classes. If one of

the conditions is true, we add V to a list of backdoors S. If a set of variables V is a backdoor, then all the subsets of variables that include V are also backdoors. We focus on finding minimal backdoors, so it is unnecessary to explore further once V is a backdoor set. Therefore, the procedure returns to the upper level with the Boolean value true. The procedure recursively calls itself with one more variable added to V and $k-1$. When $k \leq 1$, the algorithm reaches the base case, which means no backdoors have been found. Thus, the algorithm returns false.

Procedure `expand(`V, S, k`)`

input : A set of variables V
A set of backdoors S
A positive integer k
output: Return true if V is a backdoor; otherwise, return false
S is updated with new backdoors

```
foreach value assignment a_V of V do
    Propagate a_V;
    if no conflicts during propagation then
        if isSatisfied(F) or isSatisfying2SAT(F) or setValue(F, 0) or
        setValue(F, 1) then
            S ← S ∪ V;
            return true;
if k ≤ 1 then return false;
j ← index of the last variable in V;
for i ← (j + 1) to n − 1 do expand(V ∪ {x_i}, S, k − 1);
return false;
```

Because the partial assignment could lead to an unsatisfiable 2SAT formula, the procedure `isSatisfying2SAT(`F`)` is used to decide the satisfiability of the simplified 2SAT formula. The first part of the procedure is the same as `is2SAT(`F`)`, checking if there are at most two unassigned variables in each not yet satisfied clause. Next, if the remaining formula is 2SAT, we need to determine its satisfiability. Each remaining variable x_i is first assigned the value 1. If there is a conflict during unit propagation, x_i is set to 0. If a conflict occurs again during propagation, the remaining 2SAT is unsatisfiable and the algorithm returns false. If there is no conflict during propagation, we cancel the value assignments of x_i and caused by the propagation of x_i.

It can be proved that the above method can determine the satisfiability of a 2SAT formula. If a 2SAT formula F has a variable x, the clauses C of F can be divided into $\{C_1 : \text{clauses that can be solved by propagating the value of } x\}$ and $\{C_2 : \text{clauses that cannot be solved by propagating the value of } x\}$. If both values 0 and 1 of x result in conflicts during propagation, then the formula F is unsatisfiable. If propagating the value of x can satisfy the clauses in C_1, then the satisfiability of F depends on the clauses in C_2. And the satisfiability of the clauses in C_2 can be decided by propagating variables contained in C_2. The worse-case complexity is $O\left(poly(n)\, n\right)$, where $poly(n)$ is the time complexity of unit propagation. However, the exact algorithm is only

Procedure `isSatisfying2SAT(`F`)`

input : Formula F
output: Return true if the remaining formula is a satisfying 2SAT; otherwise, return false

```
foreach clause C ∈ F do
    count ← 0;
    if C is not yet satisfied then
        foreach variable x_i ∈ C do
            if x_i has not been assigned a value then count ← count + 1;
        if count > 2 then return false;
foreach variable x_i ∈ X do
    if x_i has not been assigned a value then x_i ← 1 and propagate;
    if conflicts during propagation then
        x_i ← 0 and propagate;
        if conflicts during propagation then return false;
    Cancel value assignments;
return true;
```

applied to small problem instances, so the runtime is acceptable in practice.

Many algorithms for solving 2SAT formulas have been developed in previous research. Aspvall, Plass, and Tarjan [1] propose a linear-time algorithm to solve 2SAT formulas by constructing a directed graph from the formula. We chose to use the procedure `isSatisfying2SAT(`F`)` because of its simplicity. Our purpose is to find backdoors, so we only want to decide the satisfiability, and there is no need to solve the 2SAT formula.

Example 4.1 shows how Algorithm 4.2 performs on the random 3SAT instance *uf50-01* with $k = 3$, which is the minimal backdoor size.

Example 4.1. *Figure 4.1 shows the search space. The exact algorithm explores the search states in the order of* $\{0\} \Rightarrow \{0,1\} \Rightarrow \{0,1,2\} \Rightarrow \ldots \Rightarrow \{0,1,49\} \Rightarrow \{0,2\} \Rightarrow \{0,2,3\} \Rightarrow \ldots \Rightarrow \{0,2,49\} \Rightarrow \{0,3\} \Rightarrow \ldots \Rightarrow \{48\} \Rightarrow \{48,49\} \Rightarrow \{49\}$. *The instance* uf50-01 *has eight minimal backdoors of size 3, which are* $\{2,3,37\}$, $\{2,4,37\}$, $\{2,5,37\}$, $\{2,6,37\}$, $\{2,15,37\}$, $\{2,21,37\}$, $\{2,33,37\}$, *and* $\{21,37,45\}$.

If the input value of k is larger than the minimal backdoor size, Algorithm 4.2 has the deficiency of finding backdoors that are not necessarily minimal (See Example 4.2). Hence, we repeatedly run Algorithm 4.2 with $k = 1, 2, \ldots, n-1$, until minimal backdoors are found.

Example 4.2. *Figure 4.2 shows the search space when* $k = 4$. *The set of variables* $\{2,3,37\}$ *is a minimal backdoor, so all subsets of 4 variables that include* $\{2,3,37\}$ *are also backdoors. Because Algorithm 4.2 explores the search space in a depth-first way, the search state* $\{0,2,3,37\}$ *is reached before* $\{2,3,37\}$. *Thus, the exact algorithm finds the backdoor* $\{0,2,3,37\}$, *which is not minimal, before finding the minimal backdoor* $\{2,3,37\}$.

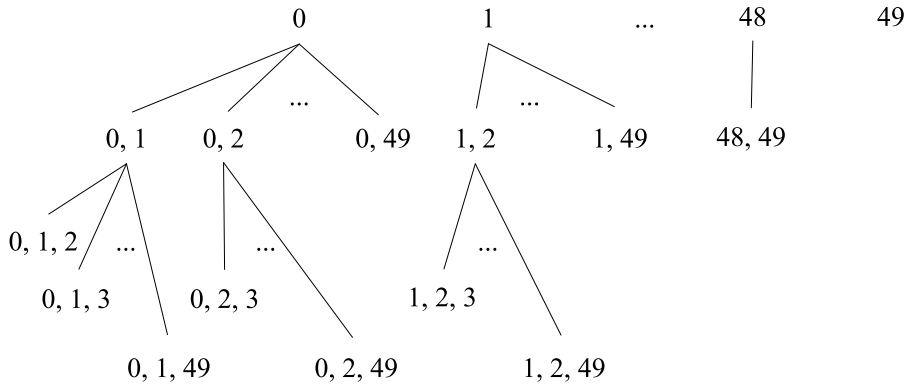

Figure 4.1: Example of the search space of Algorithm 4.2 when the input value of k equals to the minimal backdoor size

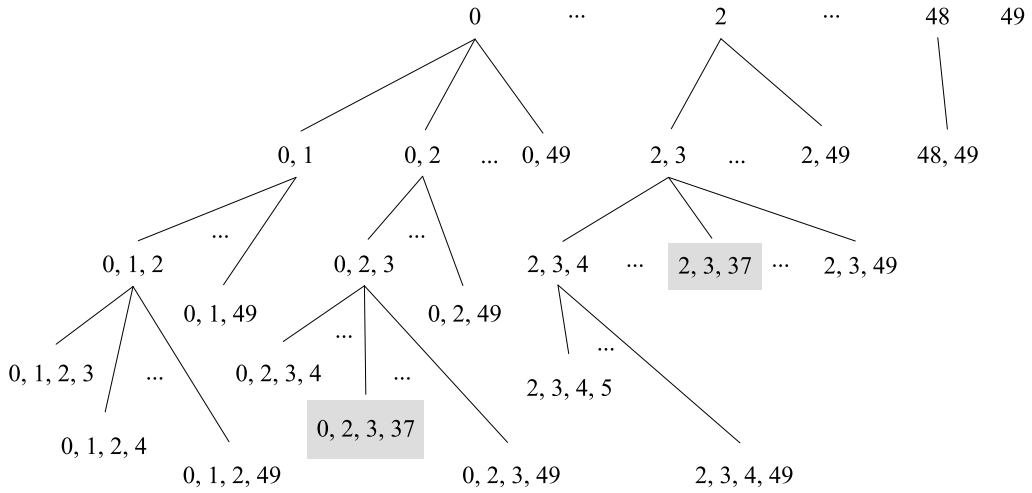

Figure 4.2: Example of the search space of Algorithm 4.2 when the input value of k is larger than the minimal backdoor size

The exact algorithm can easily be modified to find strong backdoors because it performs an exhaustive search over all subsets of k variables. The following Algorithm 4.5 finds minimal strong backdoors in unsatisfiable instances. Algorithm 4.5 calls the recursive procedure `expandStrong(`V, S, k`)`, which is a slight modification of `expand(`V, S, k`)`. If every value assignment to a set of variables V results in conflicts during unit propagation, then we are able to conclude the unsatisfiability of the instance. Thus, V is added to the list of strong backdoors S.

4.3 Local Search Algorithms

The exact algorithm based on depth-first search is one of the systematic algorithms. Although the exact algorithm is complete, it does not scale up to large problem

Algorithm 4.5: Exact algorithm for finding minimal strong backdoors in unsatisfiable instances

input : Formula F
Minimal strong backdoor size k
output: A set of minimal strong backdoors S of size k; or the empty set if F has no strong backdoors of size at most k

$S \leftarrow \emptyset$;
for $i \leftarrow 0$ **to** $n-1$ **do** `expandStrong(`$\{x_i\}$, S, k`)`;
return S;

Procedure `expandStrong(`V, S, k`)`

input : A set of variables V
A set of strong backdoors S
A positive integer k
output: Return true if V is a strong backdoor; otherwise, return false
S is updated with new strong backdoors

$count \leftarrow 0$;
foreach *value assignment* a_V *of* V **do**
  Propagate a_V;
  if *conflicts during propagation* **then** $count \leftarrow count + 1$;
if $count = 2^{|V|}$ **then**
  $S \leftarrow S \cup V$;
  return true;
if $k \leq 1$ **then return** false;
$j \leftarrow$ index of the last variable in V;
for $i \leftarrow (j+1)$ **to** $n-1$ **do** `expandStrong(`$V \cup \{x_i\}$, S, $k-1$`)`;
return false;

instances. For example, we applied the exact algorithm, Algorithm 4.2, on the *grieu-vmpc-s05-27r* instance from SAT-Race 2005 with the minimal backdoor size $k = 4$. The instance has $n = 729$ variables, but the exact algorithm did not finish within 24 hours.

Another class of search algorithms that applies here is local search. Local search algorithms are useful in solving many real-world problems. In practice, we often want to find solutions in some limited time. If systematic search algorithms are unable to complete in the limited time, no solutions can be found. In contrast, local search algorithms can return the best solutions found so far.

Our goal is to find minimal backdoors. However, local search algorithms are approximation algorithms with no guarantee of finding a backdoor with minimal size. We hope that the local search algorithms can find small backdoors that are also minimal. But it is an experimental question (addressed in Chapter 5) how often the local search algorithms find minimal backdoors.

We apply the definition of Stochastic Local Search (SLS) algorithms in Chapter 2

to the problem of backdoor detection. Given a problem instance π, the first two components of an SLS algorithm are:

- the *search space* $S(\pi)$ of instance π is the set of all subsets of variables;
- a *set of (feasible) solutions* $S'(\pi) \subseteq S(\pi)$ is the set of all backdoors;

Each search state s is a backdoor, and the evaluation function $g(s) \leftarrow |s|$ is the size of the backdoor s.

4.3.1 Previous Local Search Algorithm

Kilby, Slaney, Thiébaux, and Walsh [16] propose Algorithm 4.7, which computes small backdoors using local search.

Algorithm 4.7: Kilby *et al.*'s local search algorithm

```
input  : Formula F
         Initial backdoor set W
         Solution M
output: A set of small backdoors S

S ← ∅, B ← W;
RestartLimit ← 2, RestartCount ← 0;
IterationLimit ← √n × 3, CardMult ← 2;
while RestartCount < RestartLimit do
    RestartCount ← RestartCount + 1;
    W ← B;
    for IterationCount ← 0 to IterationLimit do
        Z ← |W| × CardMult literals chosen randomly from M \ W;
1       W ← MinWeakBackdoor(F, W ∪ Z);
2       S ← S ∪ W;
        if |W| < |B| then
            B ← W;
            RestartCount ← 0;
return S;
```

Given a formula F, Kilby *et al.* first use Satz-rand to solve F, recording the set W of branching literals and the solution M. The set W of branching literals is an initial backdoor because Satz-rand is able to solve $F[W]$ without branching. Then, Algorithm 4.7 takes the inputs F, W, and M to find small backdoors. The set B is the current smallest backdoor, which is initially the set W of branching literals. The set W is the current candidate backdoor. The algorithm has the following three constants:

- *RestartLimit*: the number of restarts;

- *IterationLimit*: the number of iterations per restart;
- $CardMult$: controlling the number of literals added to W in each iteration.

The constant *RestartLimit* is used to escape from local minima. After a number of iterations, Algorithm 4.7 restarts with the current smallest backdoor B. The constant *IterationLimit* decides how many neighbors the search explores. The constant $CardMult$ defines the neighbors of the current candidate backdoor W. In each iteration, the algorithm randomly selects from M a set Z of $|W| \times CardMult$ literals that are not in W. The set Z of literals is appended to W, and Algorithm 4.1 is called to reduce the set $W \cup Z$ of literals into a small backdoor, which is the next search state. The current candidate backdoor W is changed to the small backdoor reduced from $W \cup Z$, and then W is added to the list of backdoors S. Because Algorithm 4.1 removes literals from $W \cup Z$ sequentially, the literals of the current search state W are first removed to test whether the remaining literals form a small backdoor. If the value of $CardMult$ is too small, the set $W \cup Z$ of literals may be reduced to the same small backdoor as the current one W. If the value of $CardMult$ is too large, then the set $W \cup Z$ of literals may be reduced to a backdoor of larger size than W. Hence, it is possible that the search revisits a previous search state or moves to a worse state (See Example 4.3). If W has a smaller size than the current smallest backdoor B, B is updated with W. The values of the three constants are chosen empirically: $RestartLimit = 2$, $IterationLimit = \sqrt{n} \times 3$, and $CardMult = 2$. However, Kilby *et al.* do not provide details on how these values are decided.

The small backdoors found by Algorithm 4.7 are from one solution M. It is possible to find more small backdoors if some variables have taken different values. Therefore, Kilby *et al.* also propose an algorithm that takes as input a number of solutions randomly selected from all possible solutions. Then, Algorithm 4.7 is run repeatedly based on the chosen solutions. Kilby *et al.* use a collection of ten solutions in their experiments. Nevertheless, they do not address how to generate all possible solutions of a problem instance.

Example 4.3. *We run Algorithm 4.7 on the instance* uf50-01 *with the initial backdoor* $W = \{l_5, l_{29}, l_{44}, l_{61}, l_{68}, l_{83}, l_{93}\}$*, which is the set of variables* $\{2, 14, 22, 30, 34, 41, 46\}$*. A set* Z *of* $|W| \times CardMult = 14$ *literals randomly selected from the solution is appended to* W*. The set* $W \cup Z$ *is then reduced to* $W = \{l_{51}, l_{79}, l_{85}, l_{87}, l_{93}\}$ *by Algorithm 4.1. Thus, the set of variables* $\{25, 39, 42, 43, 46\}$ *is a small backdoor. In the next iteration, a set* Z *of* 10 *literals is appended to* W*, and then* $W \cup Z$ *is reduced to a small backdoor. The next few search states are* $\{13, 16, 26, 37\} \Rightarrow \{4, 6, 14, 25, 34, 37\} \Rightarrow \{2, 5, 37\} \Rightarrow \{2, 33, 37\} \Rightarrow \{2, 33, 37\} \Rightarrow \{23, 35, 36, 37\}$*. We can see that the search can step to a backdoor of larger size, such as the move from* $\{13, 16, 26, 37\}$ *to* $\{4, 6, 14, 25, 34, 37\}$*. The search can also step to a previously visited backdoor, such as the move from* $\{2, 33, 37\}$ *to* $\{2, 33, 37\}$*.*

In the experiments, we implement Algorithm 4.7 by modifying Minisat. As we introduced in Chapter 2, Minisat applies clause learning to solve an instance. If there is a conflict during unit propagation, implied clauses are added and learnt literals

are propagated. However, we need to turn off clause learning and use only unit propagation to find small backdoors. Therefore, constructing an initial backdoor set W requires more than adding only branching literals. We record both branching and learnt literals for the initial backdoor W. Because clause learning is turned off and no implied clauses are added, propagating the set W of branching and learnt literals does not always lead to a solution. Thus, we also add into W literals randomly picked from the solution until propagating the literals in W leads to a satisfying assignment.

4.3.2 Improvement to Previous Local Search Algorithm

Kilby *et al.* use a simple sub-solver, which applies Satz-rand's unit propagation and simplification. We modify their Algorithm 4.7 to use the more sophisticated sub-solver we define. Algorithm 4.8 is the local search algorithm after changing Line 1 and 2 of Algorithm 4.7. Line 1 is changed to call Algorithm 4.9 to find a small backdoor. We first apply Minisat's unit propagation, and then examine the simplified formula to see if it belongs to one of the polynomial-time decidable classes. We change Line 2 to add only backdoors of size less than or equal to the size of the current smallest backdoor B to the list of small backdoors S.

Algorithm 4.8: Kilby *et al.*'s local search algorithm using our proposed sub-solver

input : Formula F, Initial weak backdoor W, Solution M
output: A set of small backdoors S

$S \leftarrow \emptyset$, $B \leftarrow W$;
$RestartLimit \leftarrow 2$, $RestartCount \leftarrow 0$;
$IterationLimit \leftarrow \sqrt{n} \times 3$, $CardMult \leftarrow 2$;
while $RestartCount < RestartLimit$ **do**
    $RestartCount \leftarrow RestartCount + 1$;
    $W \leftarrow B$;
    for $IterationCount \leftarrow 0$ **to** $IterationLimit$ **do**
        $Z \leftarrow |W| \times CardMult$ literals chosen randomly from $M \setminus W$;
1       $W \leftarrow$ `findMinWeak(`F, $W \cup Z$`)`;
2   **if** $|W| \leq |B|$ **then** $S \leftarrow S \cup W$;
        if $|W| < |B|$ **then**
            $B \leftarrow W$;
            $RestartCount \leftarrow 0$;
return S;

4.3.3 Proposed Local Search Algorithm

The goal of Algorithm 4.7 is to find one small backdoor that is possibly minimal, but our goal is to find as many small backdoors as possible. Thus, based on Kilby *et*

Algorithm 4.9: `findMinWeak`(F, I)

input : Formula F, A set of literals I forming a candidate backdoor
output: A set of literals W forming a small backdoor

```
W ← ∅;
while I ≠ ∅ do
    Choose literal l ∈ I sequentially;
    I ← I \ {l};
    Run Minisat2 on F[W ∪ I];
    if Minisat2 requires branching then
        if not isSatisfied(F) and not is2SAT(F) and not setValue(F, 0)
        and not setValue(F, 1) then
            W ← W ∪ {l};
return W;
```

al.'s Algorithm 4.7, we propose Algorithm 4.10, which uses local search techniques, including:

- Tabu Search;
- best improvement strategy;
- auxiliary local search.

Algorithm 4.10: Proposed local search algorithm to find small backdoors

input : Formula F
Initial weak backdoor W
Solution M
output: A set of small backdoors S

```
curState ← findMinWeak(F, W);
preSize ← |S|, RestartLimit ← 2, RestartCount ← 0;
while RestartCount < RestartLimit do
    RestartCount ← RestartCount + 1;
    cost ← searchNeighbors(curState, S, M);
    if cost = 0 then break;
    tbList ← tbList ∪ curState;
    if |S| > preSize then RestartCount ← 0;
    preSize ← |S|;
tbList ← ∅;
localImprovement(S, M);
return S;
```

The search state *curState* is the current candidate backdoor, and *tbList* is a tabu list of previous search states. The procedure `searchNeighbors`$(curState, S, M)$ evaluates all the neighborhood of *curState* and updates *curState* with the best improving

neighbor not in *tbList*. The current state *curState* is then added to the tabu list *tbList*. The **while** loop stops if no new small backdoors have been found in the last *RestartLimit* iterations. We set *RestartLimit* to 2 in the experiments. The procedure `localImprovement`(S, M) is an auxiliary local search over the neighborhood of newly found small backdoors.

Kilby *et al.*'s Algorithm 4.7 only keeps the current search state, but does not use any memory states. We apply a tabu list to keep track of the previously visited search states. The tabu tenure is set to 30 to prevent our proposed Algorithm 4.10 from revisiting the last 30 search states. When the tabu list is full, the oldest state is replaced by the new state.

Kilby *et al.*'s Algorithm 4.7 selects the first neighbor s' encountered in the neighborhood $N(s)$ without considering its evaluation value $g(s')$. We try to improve the previous local search algorithm by choosing the best improving neighbor. Best improvement evaluates all candidate solutions in the neighborhood and selects the best improving state [13]. For a given search state s, $g^* \leftarrow \min\{g(s') \mid s' \in N(s)\}$ is the best evaluation value of all candidate solutions in the neighborhood $N(s)$. Let $I^*(s) \leftarrow \{s' \in N(s) \mid g(s') = g^*\}$ be the set of best improving neighbors. The step function can be defined as $step(s)(s') \leftarrow 1/|I^*(s)|$ if $s' \in I^*(s)$; otherwise, $step(s)(s') \leftarrow 0$. Although our proposed Algorithm 4.10 evaluates more neighbors in each iteration, the backdoor size can be reduced as much as possible.

The procedure `searchNeighbors`(W, S, M) explores all $IterationLimit$ neighbors of the current backdoor W to find a best non-tabu candidate backdoor. We set $IterationLimit$ to $\sqrt{n} \times 2$ to limit the neighborhood size in the experiments. In each iteration, a set Z of $|W| \times CardMult$ literals that are not in the current backdoor W are randomly selected from the solution M. The set $W \cup Z$ of literals is reduced to a small backdoor by Algorithm 4.9. The current candidate backdoor W is updated with this small backdoor. If W is not in the tabu list *tbList*, W is added to a list of candidate backdoors $Neighbor$. The value of *minCost* is the minimal size of backdoors in $Neighbor$. If *minCost* is no larger than the size of the current smallest backdoor, then all the backdoors in $Neighbor$ of size *minCost* are added to the list of small backdoors S. A small backdoor of size *minCost* is randomly selected from $Neighbor$ to be the next search state. When *minCost* is larger than the size of the current smallest backdoor, the search can escape from local minima by making worse moves. If every non-tabu candidate backdoor in $Neighbor$ has a larger size than the current smallest backdoor, the search moves to a best candidate backdoor from $Neighbor$.

Notice from the minimal backdoors in *uf50-01* that some variables appear in most backdoors. For example, the variable $\{37\}$ appears in all minimal backdoors. Moreover, backdoor sets $\{2, 3, 37\}$, $\{2, 4, 37\}$, $\{2, 5, 37\}$, $\{2, 6, 37\}$, $\{2, 15, 37\}$, $\{2, 21, 37\}$ and $\{2, 33, 37\}$ only differ from each other by one variable. Once a minimal backdoor s is found, it is possible to find more minimal backdoors by replacing a few variables in s. Thus, we try to find more backdoors by performing an auxiliary local search on the newly found small backdoors.

Initially we try to construct the neighborhood by replacing one variable in a small backdoor s with a variable not in s. However, this approach introduces two new

Procedure `searchNeighbors(`*W, S, M*`)`

input : Current backdoor W
A set of backdoors S
Solution M

output: Size of the next candidate backdoor W; or 0 if no candidate backdoors available
S is updated with new backdoors
W is updated with the next candidate backdoor

$IterationLimit \leftarrow \sqrt{n} \times 2$, $CardMult \leftarrow 2$;
$Neighbor \leftarrow \emptyset$, $Cost \leftarrow \emptyset$;
for $IterationCount \leftarrow 0$ **to** $IterationLimit$ **do**
  $Z \leftarrow |W| \times CardMult$ literals chosen randomly from $M \setminus W$;
  $W \leftarrow$ `findMinWeak(`F, $W \cup Z$`)`;
  if $W \notin tbList$ **then**
    $Neighbor \leftarrow Neighbor \cup W$;
    $Cost \leftarrow Cost \cup |W|$;
if $|Neighbor| = 0$ **then return** 0;
$minCost \leftarrow \min(Cost)$;
if $minCost \leq$ *current smallest backdoor size* **then**
  $S \leftarrow S \cup \{B \in Neighbor \mid |B| = minCost\}$;
$W \leftarrow$ select a backdoor from $Neighbor$ with size $minCost$ randomly;
return $minCost$;

problems. The first problem is which variable in s is selected to be replaced, and one solution is to select a backdoor variable that appears in the fewest of the clauses. The second problem is after replacing a variable in s, the new set of variables may not be a backdoor. Thus, the evaluation function $g(s) \leftarrow |s|$ does not work. One way to define a new evaluation function is:

$$g(s) \leftarrow \begin{cases} |s| & \text{if } s \text{ is a backdoor} \\ \text{the number of not yet satisfied clauses} & \text{otherwise} \end{cases}$$

However, the above mentioned methods do not seem to work well in the experiments because they require extra time for calculating the number of clauses. Therefore, we propose the procedure `localImprovement(`S, M`)`, which adds one variable to the small backdoor and calls Algorithm 4.9 to test for possible new small backdoors. For each newly found backdoor B, a literal l that is not in B is appended to B. Because the literals in B are removed sequentially by Algorithm 4.9, new backdoors can be found if the added literal l is a backdoor variable. If the size of B is smaller or equal to the size of the current smallest backdoor, then B is added to the list of small backdoors S.

Example 4.4. *We run our proposed Algorithm 4.10 on the instance* uf50-01 *with*

Procedure `localImprovement`(S, M)

input : A set of backdoors S
Solution M

foreach *new backdoor* $B \in S$ **do**
  if $B \notin tbList$ **then**
    $tbList \leftarrow tbList \cup B$;
    foreach *literal* $l \in \{M \setminus B\}$ **do**
      $B \leftarrow$ `findMinWeak`(F, $B \cup l$);
      if $|B| \leq$ *current minimum backdoor size* **then** $S \leftarrow S \cup B$;

the initial backdoor $W = \{l_5, l_{29}, l_{44}, l_{61}, l_{68}, l_{83}, l_{93}\}$*, which is the set of variables* $\{2, 14, 22, 30, 34, 41, 46\}$*. The procedure* `searchNeighbors` *selects the following search states:* $\{21, 39, 45, 46\} \Rightarrow \{21, 37, 45\} \Rightarrow \{22, 37, 45, 48\}$*. Notice that the search escapes from the local minimum* $\{21, 37, 45\}$ *by making a worse move to* $\{22, 37, 45, 48\}$*. Then the procedure* `localImprovement` *adds one variable to the backdoor* $\{21, 37, 45\}$ *and finds the backdoor* $\{2, 21, 37\}$*, which consequently leads to the detection of backdoor sets* $\{2, 3, 37\}$*,* $\{2, 4, 37\}$*,* $\{2, 5, 37\}$*,* $\{2, 6, 37\}$*,* $\{2, 15, 37\}$*, and* $\{2, 33, 37\}$*.*

The small backdoors found by the local search algorithms depend on the initial solution. We repeatedly run Minisat with successive seeds to obtain different initial solutions for the local search algorithms. However, the initial solutions do not differ from each other very much, and the sets of small backdoors found tend to be very similar.

4.4 Summary

In this chapter, I explained how we define sub-solvers both algorithmically and syntactically. I described an exact algorithm for finding minimal backdoors. I presented the local search algorithm proposed by Kilby *et al.*. I described a local search algorithm that combines our definition of sub-solvers with Kilby *et al.*'s algorithm. I also discussed the techniques used by our proposed local search algorithm.

In the next chapter, I will empirically evaluate the algorithms described in this chapter.

Chapter 5

Experimental Evaluation

In this chapter, I describe experiments on random 3SAT, structured and real-world instances to compare the five algorithms described in Chapter 4. I introduce the setup and test instances for the experiments. I also analyze the performance of the algorithms on finding weak and strong backdoors. I refer to the five algorithms by the following short names:

exact Algorithm 4.2, the exact algorithm for finding minimal weak backdoors in satisfiable instances;

strong Algorithm 4.5, the exact algorithm for finding minimal strong backdoors in unsatisfiable instances;

kilby Algorithm 4.7, the local search algorithm proposed by Kilby *et al.* [16];

kilbyImp Algorithm 4.8, the local search algorithm that incorporates our definition of sub-solvers with Algorithm 4.7;

myAlg Algorithm 4.10, our proposed local search algorithm.

5.1 Experimental Setup

We conducted experiments on various SAT instances, including uniform random 3SAT, planning, automotive configuration, and real-world instances. The set of satisfiable test instances comprises the uniform random 3SAT and planning instances from SATLIB [12], as well as the real-world instances from SAT competitions. The set of unsatisfiable test instances is from the domain of automotive product configuration [27].

We chose two test sets, *uf50-218* and *uf75-325*, of uniform random 3SAT instances. Each test set contains 100 satisfiable instances sampled from the phase transition region. Each instance in the set *uf50-218* has 50 variables and 218 clauses. Each instance in the set *uf75-325* has 75 variables and 325 clauses. The planning test set contains seven blocks world planning instances and four logistics planning instances. For real-world instances, three instances are from the SAT Competition of 2002, three

instances are from SAT-Race 2005, and the other instances are from SAT-Race 2008. The instances in previous experiments were also from domains including random 3SAT, structured, and real-world. However, most instances were small instances with fewer than 10000 variables. Minisat can solve the previous test instances quickly. We carried out experiments on larger and more difficult instances, especially the SAT-Race 2008 instances.

We selected the satisfiable instances from SAT-Race 2008. Because we implemented the algorithms based on Minisat, we did not use any instance that Minisat was unable to solve within some given time. Table 5.1 shows the reason why some satisfiable SAT-Race 2008 instances were not used in our experiments.

Table 5.1: Satisfiable SAT-Race 2008 instances not used in the experiments

Instance	**Reason**
grieu-vmpc-27	the same instance as grieu-vmpc-s05-27r from SAT-Race 2005
simon-s02b-r4b1k1.1	the same instance as simon-mixed-s02bis-01 from SAT-Race 2005
ibm-2002-30r-k85 grieu-vmpc-31 vange-col-abb313GPIA-9-c	Minisat was unable to solve these instances within the given time in SAT-Race 2008
post-cbmc-zfcp-2.8-u2 velev-npe-1.0-9dlx-b71	the process was killed when we used Minisat to simplify these instances
anbul-part-10-15-s	Minisat was unable to solve this instance within our 15-hour cutoff time for local search algorithms

First, we used Minisat to pre-process all the instances. Table 5.2 and 5.3 list for each instance the number of variables and clauses before and after Minisat's simplification. The last column is the time used by Minisat to solve the original instances. The number of clauses of the instances is greatly reduced after the simplification of Minisat. Especially for the *een-tip-sat-texas-tp-5e* instance, a total of 52128 clauses is reduced to only 153 after pre-processing.

Second, we applied the **strong** algorithm to the simplified unsatisfiable instances and the other four algorithms to the simplified satisfiable instances. The **exact** algorithm was run only on small instances to ensure reasonable runtime. With different initial solutions as inputs, the local search algorithms, **kilby**, **kilbyImp**, and **myAlg**, were run repeatedly until a cutoff time was reached. Only the minimal backdoors found by the algorithms were recorded. The experiments on real-world instances were run on the whale cluster of the Sharcnet systems (www.sharcnet.ca). Each node of the whale cluster is equipped with four Opteron CPUs at 2.2 GHz and 4.0 GB memory. The experiments on random 3SAT instances were run on a single Pentium 4 CPU at 3.2 GHz with 1 GB memory.

Table 5.2: Time used by Minisat to solve selected SAT instances

Instance	Original		Simplified		Time
	# Vars	# Clauses	# Vars	# Clauses	(seconds)
SAT Competition 2002					
apex7_gr_rcs_w5.shuffled	1500	11695	1500	11136	0.046
dp10s10.shuffled	8372	23004	8372	8557	0.507
bart11.shuffled	162	684	162	675	61.843
SAT-Race 2005					
grieu-vmpc-s05-24s	576	67872	576	49478	46.155
grieu-vmpc-s05-27r	729	96849	729	71380	606.132
simon-mixed-s02bis-01	2424	14812	2424	13793	461.450
Blocks world planning					
anomaly	48	261	48	182	0.004
medium	116	953	116	661	0.003
huge	459	7054	459	4598	0.011
bw_large.a	459	4675	459	4598	0.011
bw_large.b	1087	13772	1087	13652	0.030
bw_large.c	3016	50457	3016	50237	0.222
bw_large.d	6325	131973	6325	131607	1.296
Logistics planning					
logistics.a	828	6718	828	3116	0.024
logistics.b	843	7301	843	3480	0.022
logistics.c	1141	10719	1141	5867	0.048
logistics.d	4713	21991	4713	16588	0.101

5.2 Experiments on Finding Weak Backdoors

If the instances have small backdoors, the **exact** algorithm is sufficient to find minimal backdoors. The **exact** algorithm was applied to the random 3SAT instances. Each of the *uf50-218* and *uf75-325* test sets contains 100 instances, so we show the average number over 100 instances here. The fifth column of Table 5.4 is the average number of clauses after Minisat's simplification. We can see that the reduction in the number of clauses of random 3SAT instances is not as significant as that of real-world instances. The last column of Table 5.4 shows the average size of minimal backdoors found by the **exact** algorithm for *uf50-218* and *uf75-325*. The percentage is the ratio of the size of backdoors to the number of variables. The minimal backdoor sizes of the 100 *uf50-218* instances range from 2 to 4, while the minimal backdoor sizes of the 100 *uf75-325* instances range from 3 to 5. The average size of minimal backdoors in *uf50-218* is 3.09, which is 6.18% of the total number of variables. The average size of minimal backdoors in *uf75-325* is 3.73, which is 4.97% of the total number of variables. Although Interian [14] claims that backdoors for random 3SAT comprise 30% to 65% of the number of variables, we observe considerably smaller percentages in our experiments. Therefore, smaller backdoors can be found when features of current

SAT solvers, such as unit propagation, are considered. Gregory *et al.* [11] used exactly the same set of *uf75-325* instances in their experiments. They reported the mean backdoor size for 4 instances, and the sizes were between 4.86 to 9.96. The average backdoor size in our experiments is 3.73, which is smaller than their reported values. We found smaller backdoors because we used a systematic search algorithm, while Gregory *et al.* used an approximation algorithm to find small backdoors. Moreover, Gregory *et al.* used a DPLL-based sub-solver, but we used a complex sub-solver. Our proposed sub-solver first applies unit propagation, and then examines polynomial-time tractable classes.

We evaluated the performance of the local search algorithms, **kilby**, **kilbyImp** and **myAlg**, on random 3SAT instances. For each instance, we compared the small backdoors found by the local search algorithms in a given time period to those found by the **exact** algorithm to see if they were exactly the same. The time periods for the *uf50-218* instances were increased by 2 seconds until 10 seconds. The time periods for the *uf75-325* instances were increased by 10 seconds until 60 seconds. The percentage of instances in which the local search algorithms found the same set of backdoors as the **exact** algorithm are shown in Table 5.5 and 5.6. We notice that in a given time period, our proposed **myAlg** algorithm found more minimal backdoors than **kilby** and **kilbyImp**. The percentage of the **kilby** algorithm is the lowest because it uses a simpler sub-solver than the other algorithms.

Table 5.7 presents the results of applying the **exact** algorithm on small real-world instances. The fourth column of Table 5.7 shows the minimal backdoor size, and the last column is the number of minimal backdoors found by **exact**. We notice that the size of minimal backdoors is very small compared to the number of variables of the instances, which agrees with Williams *et al.*'s result that practical instances generally have small tractable structures [29]. The sizes of minimal backdoors in the blocks world instances are smaller than those reported by Dilkina *et al.* [7]. Dilkina *et al.* reported percentages between 1.09% to 4.17% even though they used clause learning in addition to unit propagation. We conjecture the reason is that our sub-solver not only applies unit propagation, but also tests for polynomial-time syntactic classes.

For the real-world instances, we compared the small backdoors found by the local search algorithms, **kilby**, **kilbyImp**, and **myAlg**, in a given time period. The cutoff time was set to 10800 seconds (3 hours) for small instances with fewer than 10000 variables. The cutoff time was set to 54000 seconds (15 hours) for the other large instances. Table 5.8 and 5.9 show the experimental results for 3 hours and 15 hours, respectively. For each algorithm, the first column is the size of small backdoors and the second column is the number of small backdoors found by the algorithm. I use the word *timeout* to indicate that a local search algorithm failed to find a small backdoor within the cutoff time. The *velev-vliw**, *narain-vpn-clauses-8*, and *schup-l2s-motst-2-k315* instances have a tremendous amount of variables and clauses. For these instances, the local search algorithms were unable to find a small backdoor within 15 hours. Only **myAlg** found one small backdoor in the *velev-vliw-sat-4.0-b4* instance. For each instance, I highlight the algorithm that found the smallest backdoors among the three local search algorithms. If the algorithms found small

backdoors of the same size, I highlight the algorithm that found the largest number of backdoors.

The sizes of small backdoors in the three SAT Competition 2002 instances are notably smaller than the backdoor sizes reported in [17, 22] because the algorithms in [17, 22] used syntactically defined sub-solvers.

When the cutoff time was reached, we waited for the algorithms to finish the current iteration. Because **myAlg** takes a longer amount of time to complete one iteration than **kilby** and **kilbyImp**, the time when **myAlg** found the small backdoors in some SAT-Race 2008 instances was a little longer than 54000 seconds. The longest time recorded was 168 seconds after the 15-hour cutoff time. It is possible that **kilby** and **kilbyImp** would find smaller backdoors during this leeway. Although **myAlg** takes longer time in one iteration than **kilby** and **kilbyImp**, **myAlg** is able to find a larger number of backdoors in the given time. We notice that our proposed **myAlg** always finds more backdoors than **kilby** and **kilbyImp** when the backdoor size is small. For instances that have small backdoors of size less than 10, **myAlg** found a remarkably larger amount of backdoors than **kilby** and **kilbyImp**. Note that although the *mizh-sha0** instances have backdoors of size more than 200, **myAlg** still found smaller backdoors than **kilby** and **kilbyImp**.

However, for the difficult real-world instances, our proposed **kilbyImp** always outperformed **kilby** and **myAlg** in finding small backdoors. The previous **kilby** algorithm also found the same set of small backdoors as **kilbyImp** in the *mizh-md** instances. Algorithms **kilby** and **kilbyImp** found smaller backdoors in these large real-world instances because both algorithms select the first candidate backdoor encountered. When the backdoor size and the total number of variables are large, the neighborhood of the current backdoor is too huge to be explored entirely. Thus, our proposed **myAlg** algorithm spends a long time in evaluating the neighborhood to choose the best improvement.

Williams *et al.* [29] experimented on practical instances with fewer than 10000 variables and showed that practical instances had fairly small backdoors. We extend Williams *et al.*'s result to the SAT-Race 2008 instances, which have a huge amount of variables and clauses. We notice that the SAT-Race 2008 instances have backdoors that consist of hundreds of variables. However, the backdoor size is usually less than 0.5% of the total number of variables. Thus, these large real-world instances also have comparatively small tractable structures.

We further examined three real-world instances in which the local search algorithms found small backdoors of the same size. Figure 5.1 shows the number of backdoors found by **kilby**, **kilbyImp**, and **myAlg** as the time period increased. Because the **exact** algorithm found all minimal backdoors in the *grieu-vmpc-s05-24s* instance exhaustively, we also include the **exact** algorithm in the figure of *grieu-vmpc-s05-24s* for comparison. It can be seen that on these instances our proposed **myAlg** algorithm found significantly more backdoors than **kilby** and **kilbyImp**. Moreover, **myAlg** was the quickest algorithm to find all minimal backdoors in the *grieu-vmpc-s05-24s* instance.

Table 5.10 compares the number of backdoors found by the local search algorithms, **kilby**, **kilbyImp**, and **myAlg**, during several time periods. The four instances shown

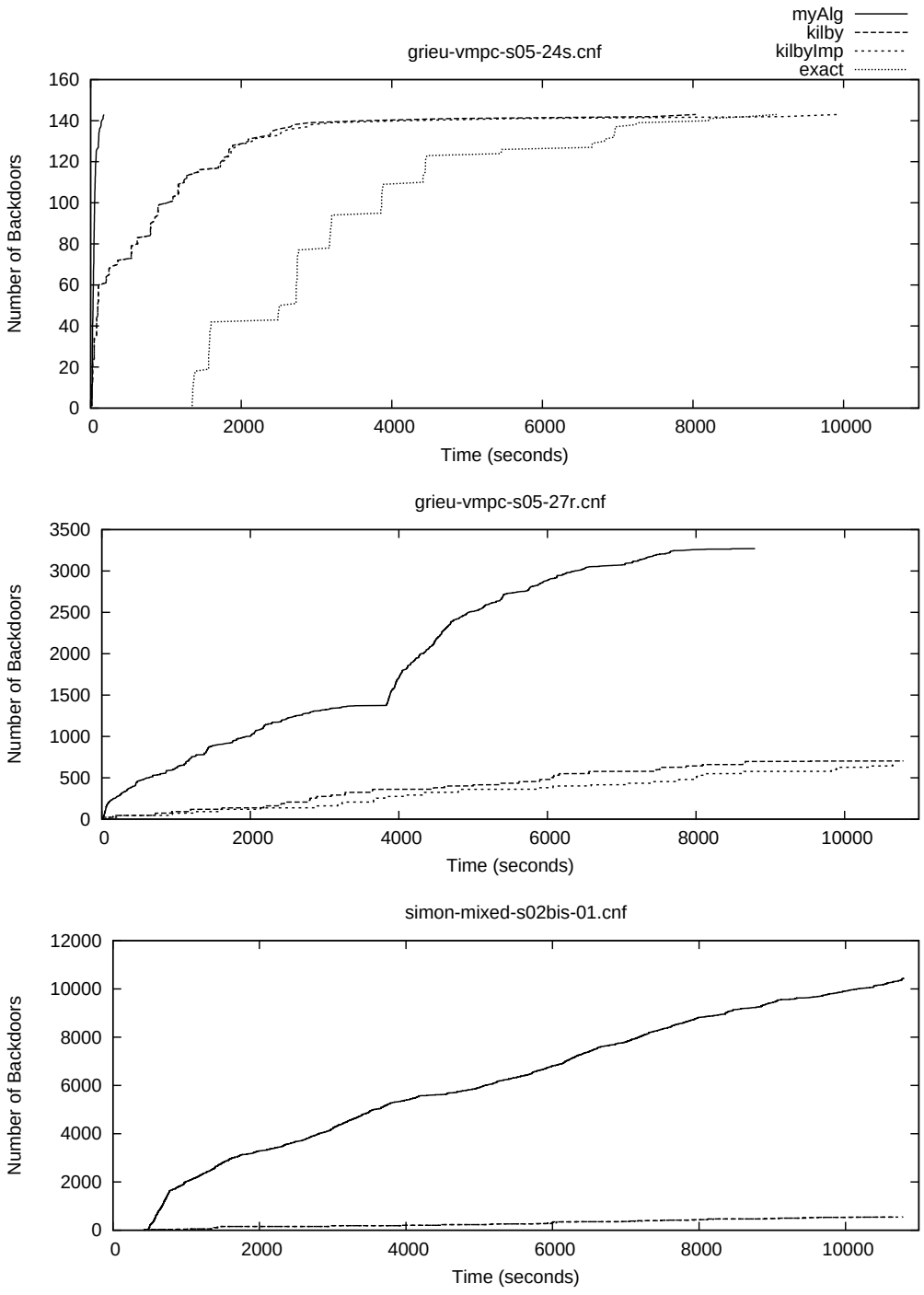

Figure 5.1: Backdoors in three real-world SAT instances

are the ones in which the local search algorithms found small backdoors of the same size. We can see that in any given time period, **myAlg** always found a larger number of backdoors than **kilby** and **kilbyImp** even though **myAlg** spent longer time to finish one iteration than the other two algorithms. By analyzing the number of backdoors found within various time periods, we could also decide how to set the cutoff time in future experiments (addressed in Chapter 6).

5.3 Experiments on Finding Strong Backdoors

In previous work [25, 6], unsatisfiable SAT benchmarks from automotive product configuration [27] were used in the experiments. Although the automotive configuration instances have about 1700 to 2000 variables, these instances are very simple. Among the 84 unsatisfiable instances, Minisat concludes the unsatisfiability of 71 instances after pre-processing. We applied the **strong** algorithm to find minimal strong backdoors for the remaining 13 instances. Results are shown in Table 5.11, where the last two columns are the size of minimal strong backdoors and the number of minimal strong backdoors. The sizes of minimal strong backdoors range from 1 to 3, which are smaller than the sizes reported in [25, 6]. We found smaller backdoors because we applied a systematic search algorithm, and we defined sub-solvers both syntactically and algorithmically.

5.4 Summary

In this chapter, I described the empirical results for the exact algorithms and the local search algorithms. The exact algorithms were sufficient to find minimal backdoors in simple instances. When large instances have relatively small backdoors, our proposed **myAlg** algorithm found considerably more backdoors than **kilby** and **kilbyImp**. The **kilbyImp** algorithm always outperformed **kilby** and **myAlg** on difficult real-world instances.

In the next chapter, I will conclude this thesis.

Table 5.3: Time used by Minisat to solve satisfiable SAT-Race 2008 instances

Instance	Original		Simplified		Time
	# Vars	# Clauses	# Vars	# Clauses	(seconds)
ibm-2002-04r-k80	104450	457628	104450	238773	300.41
ibm-2002-11r1-k45	156626	638128	156626	290625	122.88
ibm-2002-18r-k90	175216	721623	175216	370661	1566.15
ibm-2002-20r-k75	151202	623655	151202	319192	573.43
ibm-2002-22r-k75	191166	798844	191166	399095	1205.11
ibm-2002-22r-k80	203961	852379	203961	427792	2060.29
ibm-2002-23r-k90	222291	928885	222291	469900	12660.10
ibm-2002-29r-k75	64686	335189	64686	258748	1163.99
ibm-2004-01-k90	64699	276210	64699	201260	56.56
ibm-2004-1_11-k80	262808	1045990	262808	565220	1223.56
ibm-2004-23-k100	207606	861175	207606	481764	6773.58
ibm-2004-23-k80	165606	686695	165606	379170	1399.16
ibm-2004-29-k55	37714	168958	37714	123699	405.71
ibm-2004-3_02_3-k95	73525	272059	73525	169473	7.00
mizh-md5-47-3	65604	273522	65604	153650	847.47
mizh-md5-47-4	65604	273506	65604	153778	218.46
mizh-md5-47-5	65604	273520	65604	153896	507.11
mizh-md5-48-2	66892	279240	66892	157184	2730.79
mizh-md5-48-5	66892	279256	66892	157466	7676.47
mizh-sha0-35-3	48689	204067	48689	115548	458.14
mizh-sha0-35-4	48689	204067	48689	115631	1170.12
mizh-sha0-36-1	50073	210221	50073	120102	8558.69
mizh-sha0-36-3	50073	210235	50073	120212	2963.06
mizh-sha0-36-4	50073	210235	50073	120279	148.01
post-c32s-gcdm16-22	129652	386749	129652	88631	1568.71
velev-fvp-sat-3.0-b18	35853	1012271	35853	968394	7508.29
velev-vliw-sat-4.0-b4	520721	13348117	520721	13348080	2204.33
velev-vliw-sat-4.0-b8	521179	13378617	521179	13378580	3002.13
een-tip-sat-nusmv-t5.B	61933	178699	61933	42043	1.50
een-tip-sat-texas-tp-5e	17985	52128	17985	153	0.11
een-tip-sat-vis-eisen	18607	54970	18607	12801	0.48
narain-vpn-clauses-8	1461772	5687554	1461772	4572347	3496.54
palac-sn7-ipc5-h16	114548	399368	114548	218043	17164.00
palac-uts-l06-ipc5-h34	187667	874988	187667	606674	181.11
schup-l2s-motst-2-k315	507145	1601920	507145	590065	143.92
simon-s02b-r4b1k1.2	2424	14812	2424	13811	303.71
simon-s03-w08-15	132555	469519	132555	269328	145.48

Table 5.4: Minimal Backdoors for random 3SAT instances

Instance	Original		Simplified		Avg BD Size (%)
	# Vars	# Cls	# Vars	Avg # Cls	
uf50-218	50	218	50	214.88	3.09 (6.18%)
uf75-325	75	325	75	321.68	3.73 (4.97%)

Table 5.5: Comparison of the local search algorithms to the exact algorithm on uf50-218 instances

Algorithm	2s	4s	6s	8s	10s
kilby	17%	17%	17%	17%	17%
kilbyImp	87%	88%	90%	93%	97%
myAlg	90%	95%	97%	97%	98%

Table 5.6: Comparison of the local search algorithms to the exact algorithm on uf75-325 instances

Algorithm	10s	20s	30s	40s	50s	60s
kilby	9%	9%	9%	9%	9%	9%
kilbyImp	62%	75%	78%	82%	83%	85%
myAlg	79%	83%	88%	91%	91%	94%

Table 5.7: Backdoors in real-world instances found by the exact algorithm

Instance	# Vars	# Clauses	BD Size (%)	# BDs
grieu-vmpc-s05-24s	576	49478	3 (0.521%)	143
een-tip-sat-texas-tp-5e	17985	153	1 (0.006%)	2
anomaly	48	182	1 (2.083%)	2
medium	116	661	1 (0.862%)	5
huge	459	4598	2 (0.436%)	89
bw_large.a	459	4598	2 (0.436%)	89
bw_large.b	1087	13652	2 (0.184%)	7

Table 5.8: Comparison of the local search algorithms on selected SAT instances (3 hours)

Instance	# Vars	# Cls	kilby		kilbyImp		myAlg	
			BD Size (%)	# BDs	BD Size (%)	# BDs	BD Size (%)	# BDs
SAT Competition 2002								
apex7_gr_rcs_w5.shuffled	1500	11136	77 (5.133%)	1	**47 (3.133%)**	**4**	53 (3.533%)	42885
dp10s10.shuffled	8372	8557	9 (0.108%)	10520	9 (0.108%)	9573	**9 (0.108%)**	**59399**
bart11.shuffled	162	675	15 (9.259%)	4190	14 (8.642%)	2903	**14 (8.642%)**	**45044**
SAT-Race 2005 and 2008								
grieu-vmpc-s05-24s	576	49478	3 (0.521%)	143	3 (0.521%)	143	3 (0.521%)	143
grieu-vmpc-s05-27r	729	71380	4 (0.549%)	710	4 (0.549%)	660	**4 (0.549%)**	**3271**
simon-mixed-s02bis-01	2424	13793	8 (0.330%)	566	8 (0.330%)	566	**8 (0.330%)**	**10440**
simon-s02b-r4b1k1.2	2424	13811	8 (0.330%)	394	7 (0.289%)	3	**7 (0.289%)**	**16**
Blocks world planning								
bw_large.c	3016	50237	4 (0.133%)	1934	**3 (0.099%)**	**15**	**3 (0.099%)**	**15**
bw_large.d	6325	131607	6 (0.095%)	790	**5 (0.079%)**	**69**	6 (0.095%)	640
Logistics planning								
logistics.a	828	3116	20 (2.415%)	147	**20 (2.415%)**	**6675**	24 (2.899%)	584257
logistics.b	843	3480	16 (1.898%)	1688	**15 (1.779%)**	**9789**	16 (1.898%)	7634
logistics.c	1141	5867	26 (2.279%)	18	**25 (2.191%)**	**387**	28 (2.454%)	424467
logistics.d	4713	16588	25 (0.530%)	39	**22 (0.467%)**	**61**	28 (0.594%)	36610

Table 5.9: Comparison of the local search algorithms on SAT-Race 2008 instances (15 hours)

Instance	# Vars	# Cls	kilby		kilbyImp		myAlg	
			BD Size (%)	# BDs	BD Size (%)	# BDs	BD Size (%)	# BDs
ibm-2002-04r-k80	104450	238773	252 (0.241%)	10	**154 (0.147%)**	**53**	184 (0.176%)	2
ibm-2002-11r1-k45	156626	290625	307 (0.196%)	3	**282 (0.180%)**	**7**	344 (0.220%)	2
ibm-2002-18r-k90	175216	370661	360 (0.205%)	3	**331 (0.189%)**	**6**	496 (0.283%)	1
ibm-2002-20r-k75	151202	319192	319 (0.211%)	4	**275 (0.182%)**	**17**	384 (0.254%)	1
ibm-2002-22r-k75	191166	399095	453 (0.237%)	4	**424 (0.222%)**	**3**	551 (0.288%)	2
ibm-2002-22r-k80	203961	427792	499 (0.245%)	1	**466 (0.228%)**	**4**	605 (0.297%)	1
ibm-2002-23r-k90	222291	469900	537 (0.242%)	2	**534 (0.240%)**	**1**	624 (0.281%)	2
ibm-2002-29r-k75	64686	258748	81 (0.125%)	11	**58 (0.090%)**	**26**	59 (0.091%)	1
ibm-2004-01-k90	64699	201260	148 (0.229%)	2	**87 (0.134%)**	**5**	93 (0.144%)	8
ibm-2004-1_11-k80	262808	565220	696 (0.265%)	4	**648 (0.247%)**	**1**	732 (0.279%)	1
ibm-2004-23-k100	207606	481764	524 (0.252%)	2	**455 (0.219%)**	**1**	618 (0.298%)	4
ibm-2004-23-k80	165606	379170	465 (0.281%)	2	**441 (0.266%)**	**1**	550 (0.332%)	1
ibm-2004-29-k55	37714	123699	67 (0.178%)	16	52 (0.138%)	21	**49 (0.130%)**	**6381**
ibm-2004-3_02_3-k95	73525	169473	1297 (1.764%)	1	**238 (0.324%)**	**2**	251 (0.341%)	1
mizh-md5-47-3	65604	153650	**179 (0.273%)**	**1**	**179 (0.273%)**	**1**	265 (0.404%)	4
mizh-md5-47-4	65604	153778	**184 (0.280%)**	**2**	190 (0.290%)	1	232 (0.354%)	2
mizh-md5-47-5	65604	153896	**181 (0.276%)**	**2**	**181 (0.276%)**	**2**	235 (0.358%)	1
mizh-md5-48-2	66892	157184	**203 (0.303%)**	**1**	**203 (0.303%)**	**1**	289 (0.432%)	1
mizh-md5-48-5	66892	157466	**189 (0.283%)**	**6**	**189 (0.283%)**	**6**	238 (0.356%)	1
mizh-sha0-35-3	48689	115548	258 (0.530%)	1	254 (0.522%)	2	**238 (0.489%)**	**1**
mizh-sha0-35-4	48689	115631	237 (0.487%)	1	237 (0.487%)	1	**210 (0.431%)**	**1**
mizh-sha0-36-1	50073	120102	261 (0.521%)	1	261 (0.521%)	1	**219 (0.437%)**	**1**
mizh-sha0-36-3	50073	120212	249 (0.497%)	1	260 (0.519%)	4	**209 (0.417%)**	**5**
mizh-sha0-36-4	50073	120279	237 (0.473%)	1	237 (0.473%)	1	**220 (0.439%)**	**1**
post-c32s-gcdm16-22	129652	88631	12 (0.009%)	133	12 (0.009%)	133	**11 (0.008%)**	**126**
velev-fvp-sat-3.0-b18	35853	968394	228 (0.636%)	3	**212 (0.591%)**	**1**	227 (0.633%)	1
velev-vliw-sat-4.0-b4	520721	13348080	timeout		timeout		**933 (0.179%)**	**1**
velev-vliw-sat-4.0-b8	521179	13378580	timeout		timeout		timeout	
een-tip-sat-nusmv-t5.B	61933	42043	109 (0.176%)	6	**88 (0.142%)**	**35**	92 (0.149%)	14318
een-tip-sat-vis-eisen	18607	12801	8 (0.043%)	6087	8 (0.043%)	16466	**8 (0.043%)**	**36941**
narain-vpn-clauses-8	1461772	4572347	timeout		timeout		timeout	
palac-sn7-ipc5-h16	114548	218043	10 (0.009%)	46	10 (0.009%)	46	**10 (0.009%)**	**1533**
palac-uts-l06-ipc5-h34	187667	606674	**10 (0.005%)**	**152**	**10 (0.005%)**	**152**	10 (0.005%)	102
schup-l2s-motst-2-k315	507145	590065	timeout		timeout		timeout	
simon-s03-w08-15	132555	269328	233 (0.176%)	26	**115 (0.087%)**	**31**	152 (0.115%)	4

Table 5.10: The number of backdoors found by the local search algorithms for various time periods

Instance	3600s			7200s			10800s		
	kilby	kilbyImp	myAlg	kilby	kilbyImp	myAlg	kilby	kilbyImp	myAlg
dp10s10.shuffled	3943	3576	19716	7457	6948	39768	10520	9573	59399
grieu-vmpc-s05-24s	139	139	143	141	141	143	143	143	143
grieu-vmpc-s05-27r	324	206	1372	577	434	3118	710	660	3271
simon-mixed-s02bis-01	185	185	5019	395	379	8020	566	566	10440

Table 5.11: Minimal strong backdoors in the car configuration instances

Instance	Original		Simplified		BD Size (%)	# BDs
	# Vars	# Cls	# Vars	# Cls		
C168_FW_SZ_128	1698	5425	1698	119	3 (0.177%)	6
C168_FW_SZ_66	1698	5401	1698	60	1 (0.059%)	3
C202_FS_RZ_44	1750	6199	1750	119	2 (0.114%)	26
C202_FW_SZ_87	1799	8946	1799	305	3 (0.167%)	90
C210_FS_RZ_23	1755	5778	1755	161	3 (0.171%)	17
C210_FS_RZ_38	1755	5763	1755	118	2 (0.114%)	4
C210_FS_SZ_103	1755	5775	1755	280	2 (0.114%)	3
C210_FW_RZ_30	1789	7426	1789	338	3 (0.168%)	16
C210_FW_RZ_57	1789	7405	1789	330	2 (0.112%)	4
C210_FW_SZ_106	1789	7417	1789	555	2 (0.112%)	3
C210_FW_SZ_128	1789	7412	1789	447	1 (0.056%)	3
C210_FW_UT_8630	2024	9721	2024	577	1 (0.050%)	2
C220_FV_SZ_65	1728	4496	1728	85	1 (0.058%)	2

Chapter 6

Conclusion and Future Work

In this chapter, I summarize the work presented in this thesis and discuss possible future work.

6.1 Summary of the Thesis

Williams, Gomes, and Selman [29] introduce the concept of backdoors to explain why state-of-the-art solvers scale well on large practical instances. The interest of our research is finding all minimal backdoors in SAT instances because we want to study how value and variable ordering mistakes affect the performance of backtracking algorithms. Kilby, Slaney, Thiébaux, and Walsh [16] propose a local search algorithm for finding small backdoors. As well, several algorithms for backdoor detection have been proposed in previous work and experimental studies have been carried out to evaluate these algorithms.

In this thesis, I introduced our proposed definition of sub-solvers. After applying unit propagation, the sub-solver checks if the formula is already satisfied, or if the simplified formula belongs to Schaefer's polynomial-time tractable classes [26]. I presented an exact algorithm for finding all minimal weak backdoors of a certain size in satisfiable instances. Then I discussed how we modified the exact algorithm to find minimal strong backdoors in unsatisfiable instances. Based on Kilby *et al.*'s local search algorithm **kilby**, I described our proposed local search algorithms **kilbyImp** and **myAlg**. The **kilbyImp** algorithm incorporates our definition of sub-solvers with the previous **kilby** algorithm. The **myAlg** algorithm applies local search strategies, including best improvement, Tabu Search, and an auxiliary local search, to find many small backdoors. I modified Minisat to implement these algorithms and empirically evaluated the algorithms on random 3SAT, structured, and real-world instances. Experimental results showed that the size of backdoors was relatively small compared to the number of variables. The ratio of the backdoor size to the number of variables was larger for random 3SAT instances (around 5–6%) than for real-world instances (generally less than 0.5%). Although the real-world instances have a large number of variables and backdoors, they could be solved efficiently by backtracking on the variables in the small backdoors. The exact algorithms were competent to find all

minimal backdoors in the small instances. We applied local search algorithms to find small backdoors in large real-world instances. When the real-world instances had backdoors of size less than ten, our proposed **myAlg** found a substantially larger amount of small backdoors than **kilby** and **kilbyImp**. For the real-world instances with a huge number of variables and clauses, our proposed **kilbyImp** almost always found backdoors of the smallest size among the three local search algorithms.

6.2 Future Work

- The performance of the local search algorithms depends on the initial solution. If the initial solutions vary a lot from each other, the local search algorithms have a better chance of finding more new backdoors in each run. In the experiments, we ran Minisat with successive seeds to get initial solutions, but the solutions found tended to be very similar. Thus, the local search algorithms found almost the same set of backdoors in each run. It is necessary to study how to obtain a number of different solutions to an instance.

- The size of backdoors found by our proposed algorithms could be reduced if we could efficiently identify more polynomial-time tractable classes, such as Class 6 of Schaefer's theorem [26] and RHorn. As introduced in Chapter 4, previous work [15, 18] developed polynomial-time algorithms for identifying these two tractable classes. Because the algorithms require transforming the original problem into a new problem, we did not implement the algorithms proposed in [15, 18]. Future work could involve implementing these algorithms and developing new algorithms for checking polynomial-time tractable classes.

- The local search algorithms could be improved by incorporating other local search techniques, such as Randomized Iterative Improvement and Variable Neighborhood Descent [13]. Randomized Iterative Improvement extends the Iterative Improvement algorithm by choosing random neighbors with a certain probability so that worsening moves are allowed. Our proposed **myAlg** applies the best improvement strategy, which evaluates all the neighborhood and selects the best improving state. However, **myAlg** does not perform well on instances with huge neighborhoods because of its high time complexity. Variable Neighborhood Descent attempts to reduce the time complexity by changing the neighborhood as the search progresses. In the beginning, the search explores small neighborhoods until reaching a local minimum. Then the search changes to evaluating larger neighborhoods.

- The cutoff time for the local search algorithms could be formally decided. We empirically set the cutoff time to 3 hours and 15 hours in our experiments. As the time increases, the local search algorithms may find fewer backdoors in each additional time period. Thus, future work could involve the study of how to properly determine the cutoff time for the local search algorithms.

- Clause learning could be taken into account for finding minimal backdoors. Dilkina, Gomes, and Sabharwal [7] introduced the concept of learning-sensitive backdoors. We did not consider clause learning because it could increase the time complexity of the algorithms for finding backdoors. If clause learning is taken into account for finding backdoors, we need to examine both original and learnt clauses when checking polynomial-time tractable classes. Moreover, the order of value assignments to backdoor variables is important when clause learning is involved [7].

- Our work focuses on finding weak backdoors, so our proposed **strong** algorithm for finding strong backdoors is very simple and only works for unsatisfiable instances. The **strong** algorithm could be improved by considering the tests for polynomial-time tractable classes. The **strong** algorithm could also be extended to detect strong backdoors in both satisfiable and unsatisfiable instances.

- Finding all minimal backdoors is important for studying value and variable ordering mistakes. For example, a variable ordering mistake could be selecting a variable not in the backdoor. A value ordering mistake could be assigning the backdoor variable a value that does not lead to a polynomial sub-problem. Future work could be a formal definition of value and variable ordering mistakes. We could also investigate how value and variable ordering mistakes affect the performance of backtracking algorithms.

Appendix A

The Example Instance

I used the random 3SAT instance *uf50-01* as an example in Chapter 4. The instance originally has 50 variables and 218 clauses. After Minisat's simplification, the number of clauses is reduced to 215. Table A.1 shows all 215 clauses of *uf50-01* after the simplification. For a variable x_i, $i = 0, 1, \ldots, 49$, the positive literal is denoted by $(i+1)$ and the negative literal is denoted by $-(i+1)$. For example, (-3 7 36) represents the clause $(\neg x_2 \vee x_6 \vee x_{35})$. The exact algorithm found eight minimal weak backdoors of size 3 in the instance. Given the following value assignments to the backdoor variables, the *uf50-01* instance can be solved by our proposed sub-solver.

Weak Backdoors	**Value Assignments**
$\{x_2, x_3, x_{37}\}$	0, 1, 0
$\{x_2, x_4, x_{37}\}$	0, 1, 0
$\{x_2, x_5, x_{37}\}$	0, 1, 0
$\{x_2, x_6, x_{37}\}$	0, 1, 0
$\{x_2, x_{15}, x_{37}\}$	0, 0, 0
$\{x_2, x_{21}, x_{37}\}$	0, 0, 0
$\{x_2, x_{33}, x_{37}\}$	0, 0, 0
$\{x_{21}, x_{37}, x_{45}\}$	0, 0, 0

Table A.1: The random 3SAT instance *uf50-01* after Minisat's simplification

-3 7 36	-3 -42 -48	-41 -47 -49	8 17 -40	-21 -31 -39
-22 36 49	14 27 38	6 15 -18	6 7 -43	-7 23 34
2 -42 47	3 -33 -35	40 44 49	31 36 50	-3 -36 -37
26 -29 43	15 29 -45	-11 18 24	6 -26 -47	6 16 32
-34 37 41	7 -17 -28	19 -44 46	7 22 -48	3 34 39
31 -43 46	23 -27 32	-18 37 -50	5 11 20	6 -24 -45
-14 -23 -34	20 21 -22	-17 24 50	3 21 35	-26 -36 47
-28 -45 49	-6 12 -21	-15 -17 -39	2 -14 41	-23 25 36
-3 -39 -40	20 35 50	27 31 -39	-15 -40 45	34 35 50
-1 12 -48	18 -30 -35	-24 -25	-4 -12 -33	-24 -37 -43
31 -37 -44	-9 14 -38	-16 33 34	4 -5 -35	-3 -19 -21
-29 -35 -36	7 36 -43	14 30 41	-7 -24 -35	6 35 -42
-1 -15 39	-16 27 49	-3 -46 50	20 34 -41	-1 23 28
-12 -20 -30	-24 29 -37	5 12 -44	-2 -6 48	-2 -43 -49
1 24 -50	-7 -44 -50	4 -41 43	-3 -11 23	33 41 48
9 23 -49	1 -43 47	16 -29 -40	3 19 30	19 -34 48
14 -16 -44	-12 38 -45	-4 -14 -31	-1 35 -48	-7 9 42
-1 8 -15	-31 -37 -43	-27 -29 47	4 7 17	17 20 -25
-5 35 -42	-5 24 -50	2 -21 -26	-8 -21 45	-16 33 49
6 16 -38	5 21 37	8 31 38	14 -21 33	-5 20 40
-9 -29 31	-7 -22 42	8 26 -48	33 -38 48	-34 46 49
-14 25 -46	4 18 -46	-12 -31 36	12 14 -18	-7 -16 46
7 -8 9	-22 -42 49	-15 22 38	34 -41 47	22 -26 32
-21 -25 -45	-11 -26 32	15 -25 26	-1 25 46	-14 30 -31
-9 12 -22	-18 26 -35	-16 -21 -32	-21 31 -49	9 11 41
3 -4 -22	-18 -25 -50	4 9 -40	20 37 46	22 -27 -29
3 14 34	3 20 -31	2 -26 -50	17 -29 38	12 -41 -49
15 -35 -43	-22 -23 -49	-9 33 48	26 29 35	27 37 -50
-7 -43 46	-8 -37 -46	-24 36 -40	15 -44 46	-3 -16 36
9 43 -48	-4 -25 44	-7 -22 37	-17 -22 -31	-11 17 -48
23 -28 34	23 -39 -48	-1 -23 -37	14 -19 27	-6 -22 33
-6 -26 -32	18 -20 -46	22 27 43	3 -35 -46	32 39 -43
6 -9 -39	-16 27 39	-15 -17 25	27 34 -43	5 -6 49
11 14 -38	-38 40 47	-14 17 37	29 36 39	1 -28 -39
14 -16 -18	15 -40 50	18 37 -42	2 33 -42	-3 8 -22
1 23 -31	-20 26 -45	11 42 49	11 29 -43	-20 -21 30
23 -35 45	-14 -30 38	-9 -29 48	11 -18 -23	-1 -29 -41
5 26 41	-7 -30 44	-6 38 -41	-15 46 48	-32 38 46
12 -32 46	14 31 40	2 -18 49	27 28 -38	14 -16 -21
12 15 -29	5 34 49	-12 14 22	20 30 33	22 -24
4 -23 -48	9 -30 -36	12 -35 44	3 -21 38	-11 33 49
7 -33 35 -50	7 35 -37 49	4 6 7 35	7 -18 35 -47	2 -11 14 15
-11 15 19 45	-11 -14 15 -44	-11 15 19 -30	-11 15 34 49	-11 15 31 33

Bibliography

[1] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[2] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. ACM*, 5:394–397, 1962.

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7:201–215, 1960.

[4] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[5] B. Dilkina, C. P. Gomes, Y. Malitsky, A. Sabharwal, and M. Sellmann. Backdoors to combinatorial optimization: Feasibility and optimality. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-09)*, pages 56–70, Pittsburgh, Pennsylvania, USA, 2009.

[6] B. Dilkina, C. P. Gomes, and A. Sabharwal. Tradeoffs in the complexity of backdoor detection. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, pages 256–270, Providence, Rhode Island, 2007. Available as: Springer Lecture Notes in Computer Science 4741.

[7] B. Dilkina, C. P. Gomes, and A. Sabharwal. Backdoors in the context of learning. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, pages 73–79, Swansea, UK, 2009.

[8] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[9] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (Selected and Revised Papers of SAT 2003)*, pages 502–518, 2004. Available as: Springer Lecture Notes in Computer Science 2919.

[10] C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 431–437, Madison, Wisconsin, USA, 1998.

[11] P. Gregory, M. Fox, and D. Long. A new empirical study of weak backdoors. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP 2008)*, pages 618–623, Syndey, Australia, 2008.

[12] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT 2000*, pages 283–292. IOS Press, 2000. Available online at www.satlib.org.

[13] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier / Morgan Kaufmann, 2004.

[14] Y. Interian. Backdoor sets for random 3-SAT. Informal Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), 2003.

[15] P. Jeavons, D. Cohen, and M. Gyssens. Closure properties of constraints. *J. ACM*, 44(4):527–548, 1997.

[16] P. Kilby, J. Slaney, S. Thiébaux, and T. Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 1368–1373, Pittsburgh, Pennsylvania, USA, 2005.

[17] S. Kottler, M. Kaufmann, and C. Sinz. Computation of renameable horn backdoors. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT 2008)*, pages 154–160, Guangzhou, China, 2008.

[18] H. R. Lewis. Renaming a set of clauses as a Horn set. *J. ACM*, 25:134–135, 1978.

[19] C. M. Li and A. Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 366–371, Nagoya, Japan, 1997.

[20] D. G. Mitchell. A SAT solver primer. *EATCS Bulletin (The Logic in Computer Science Column)*, 85:112–133, 2005.

[21] N. Nishimura, P. Ragde, and S. Szeider. Detecting backdoor sets with respect to Horn and binary clauses. In *Proceedings of Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 96–103, Vancouver, Canada, 2004.

[22] L. Paris, R. Ostrowski, P. Siegel, and L. Saïs. Computing Horn strong backdoor sets thanks to local search. In *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2006)*, pages 139–143, Washington, DC, USA, 2006.

[23] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[24] Y. Ruan, H. Kautz, and E. Horvitz. The backdoor key: A path to understanding problem hardness. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 124–130, San Jose, California, USA, 2004.

[25] M. Samer and S. Szeider. Backdoor trees. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI-08)*, pages 363–368, Chicago, USA, 2008.

[26] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing (STOC'78)*, pages 216–226, New York, NY, USA, 1978.

[27] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17(1):75–97, 2003.

[28] S. Szeider. Backdoor sets for DLL subsolvers. *Journal of Automated Reasoning*, 35:73–88, 2005.

[29] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1173–1178, Acapulco, Mexico, 2003.

[30] R. Williams, C. Gomes, and B. Selman. On the connections between backdoors and heavy-tails on combinatorial search. In *Proceedings of SAT 2003*, Santa Margherita Ligure, Italy, 2003.